UNCLASSIFIED

AD NUMBER

ADB120259

LIMITATION CHANGES

TO:

Approved for public release; distribution is unlimited.

FROM:

Distribution authorized to U.S. Gov't. agencies and their contractors; Critical Technology; MAR 1988. Other requests shall be referred to Air Force Armament Lab., Eglin AFB, FL. This document contains export-controlled technical data.

AUTHORITY

AFSC/MNOL Wright Lab ltr dtd 13 Feb 1992

THIS PAGE IS UNCLASSIFIED

DTIC FILE COPY

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1a. REPORT SECURITY CLASSIFICATION Unclassified	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT Distribution authorized to U.S. Government Agencies and their contractors; CT (over)
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S)	5. MONITORING ORGANIZATION REPORT NUMBER(S) AFATL-TR-88-18, Vol 12
6a. NAME OF PERFORMING ORGANIZATION McDonnell Douglas Astronautics Company 6b. OFFICE SYMBOL (If applicable)	7a. NAME OF MONITORING ORGANIZATION Aeromechanics Division
6c. ADDRESS (City, State, and ZIP Code) P.O. Box 516 St. Louis, MO 63166	7b. ADDRESS (City, State, and ZIP Code) Air Force Armament Laboratory Eglin AFB, FL 32542-5434
8a. NAME OF FUNDING/SPONSORING ORGANIZATION STARS Joint Program Office 8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER F08635-86-C-0025
8c. ADDRESS (City, State, and ZIP Code) Room 3D139 (1211 Fern St) The Pentagon Washington DC 20301-3081	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. 63756D; PROJECT NO. 921C; TASK NO. GZ; WORK UNIT ACCESSION NO. 57

11. TITLE (Include Security Classification)
Common Ada Missile Package (CAMP) Project: Missile Software Parts, Vol 12: Detail Design Documents (Vol 7-12)

12. PERSONAL AUTHOR(S)
D. McNicholl, S. Cohen, C. Palmer, et al.

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Technical Note	FROM Sep 85 TO Mar 88	March 1988	230

16. SUPPLEMENTARY NOTATION
SUBJECT TO EXPORT CONTROL LAWS.
Availability of this report is specified on verso of front cover. (over).

17. COSATI CODES		
FIELD	GROUP	SUB-GROUP

18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
Reusable Software, Missile Software, Software Generators
Ada, Parts Composition Systems, Software Parts

19. ABSTRACT (Continue on reverse if necessary and identify by block number)
The objective of the CAMP program is to demonstrate the feasibility of reusable Ada software parts in a real-time embedded application area; the domain chosen for the demonstration was that of missile flight software systems. This required that the existence of commonality within that domain be verified (in order to justify the development of parts for that domain), and that software parts be designed which address those areas identified. An associated parts system was developed to support parts usage. Volume 1 of this document is the User's Guide to the CAMP Software parts; Volume 2 is the Version Description Document; Volume 3 is the Software Product Specification; Volumes 4-6 contain the Top-Level Design Document; and, Volumes 7-12 contain the Detail Design Documents.

part of

DTIC ELECTE APR 07 1988 S D E

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT ☐ UNCLASSIFIED/UNLIMITED ☒ SAME AS RPT. ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION Unclassified
22a. NAME OF RESPONSIBLE INDIVIDUAL Christine Anderson	22b. TELEPHONE (Include Area Code) (904) 882-2961 22c. OFFICE SYMBOL AFATL/FXG

DD Form 1473, JUN 86 — Previous editions are obsolete. — SECURITY CLASSIFICATION OF THIS PAGE
UNCLASSIFIED

AD-B120 259

3. DISTRIBUTION/AVAILABILITY OF REPORT (CONCLUDED)

~~this report documents test and evaluation~~; distribution limitation applied March 1988. Other requests for this document must be referred to AFATL/FXG, Eglin AFB, Florida 32542-5434.

16. SUPPLEMENTARY NOTATION (CONCLUDED)

These technical notes accompany the CAMP final report AFATL-TR-85-93 (3 Vols)

AFATL-TR-88-18, Vol 12

SOFTWARE DETAILED DESIGN DOCUMENT

FOR THE

MISSILE SOFTWARE PARTS

OF THE

COMMON ADA MISSILE PACKAGE (CAMP) PROJECT

CONTRACT F08635-86-C-0025

CDRL SEQUENCE NO. C007



Accession For
NTIS GRA&I ☐
DTIC TAB ☒
Unannounced ☐
Justification
By
Distribution/
Availability Codes
Dist | Avail and/or Special
C-2

30 OCTOBER 1987

Distribution authorized to U.S. Government agencies and their contractors only; ~~this report documents test and evaluation;~~ distribution limitation applied July 1987. Other requests for this document must be referred to the Air Force Armament Laboratory (FXG) Eglin Air Force Base, Florida 32542-5434.

DESTRUCTION NOTICE - For classified documents, follow the procedures in DoD 5220.22-M, Industrial Security Manual, Section II-19 or DoD 5200.1-R, Information Security Program Regulation, Chapter IX. For unclassified, limited documents, destroy by any method that will prevent disclosure of contents or reconstruction of the document.

WARNING: This document contains technical data whose export is restricted by the Arms Export Control Act (Title 22, U.S.C., Sec. 2751, et seq.) or the Export Administration Act of 1979, as amended (Title 50, U.S.C., App. 2401, et seq.). Violations of these export laws are subject to severe criminal penalties. Disseminate in accordance with the provisions of AFR 80-34.

AIR FORCE ARMAMENT LABORATORY
Air Force Systems Command ▮ United States Air Force ▮ Eglin Air Force Base, Florida

88 4 6 131

3.3.7 ABSTRACT MECHANISMS

(This page intentionally left blank.)

3.3.7.1 ABSTRACT_DATA_STRUCTURES TLCSC P691 (CATALOG #P330-0)

This package contains the bodies of the generic packages required to define and manipulate the following abstract data structures:

o bounded FIFO buffer o unbounded FIFO buffer o nonblocking circular buffer o unbounded priority queue o bounded stack o unbounded stack

It also contains the package required by the unbounded parts to handle the manipulation of their available space lists.

The decomposition for this part is the same as that shown in the Top-Level Design Document.

3.3.7.1.1 REQUIREMENTS ALLOCATION

The following chart summarizes the allocation of CAMP requirements to this part:

Name	Requirements Allocation
Bounded_FIFO_Buffer	R125
Unbounded_FIFO_Buffer	R164
Nonblocking_Circular_Buffer	R126
Unbounded_Priority_Queue	R165
Bounded_Stack	R166
Unbounded_Stack	R167

3.3.7.1.2 LOCAL ENTITIES DESIGN

Packages:

The following table describes the packages maintained local to this part:

Name	Type	Description
Available_Space_List_Operations	generic package	Contains a set of functions to retrieve a node from and add a node to an available space list

3.3.7.1.3 INPUT/OUTPUT

None.

3.3.7.1.4 LOCAL DATA

None.

3.3.7.1.5 PROCESS CONTROL

Not applicable.

3.3.7.1.6 PROCESSING

The following describes the processing performed by this part:

```
package body Abstract_Data_Structures is

-- -------------------------
-- --separate package bodies
-- -------------------------

   package body Bounded_FIFO_Buffer is separate;

   package body Unbounded_FIFO_Buffer is separate;

   package body Nonblocking_Circular_Buffer is separate;

   package body Unbounded_Priority_Queue is separate;

   package body Bounded_Stack is separate;

   package body Unbounded_Stack is separate;

   package body Available_Space_List_Operations is separate;

end Abstract_Data_Structures;
```

3.3.7.1.7 UTILIZATION OF OTHER ELEMENTS

None.

3.3.7.1.8 LIMITATIONS

None.

3.3.7.1.9 LLCSC DESIGN

3.3.7.1.9.1 AVAILABLE_SPACE_LIST_OPERATIONS PACKAGE DESIGN

This package contains a set of routines used to manipulate an available space list which is maintained local to the part instantiating this package.

The first routine, New_Node, will return a node to the calling routine. If a node is available in the available space list, the node will be retrieved from there. If not, a new node will be dynamically allocated. If no memory is available for the allocation, a STORAGE_ERROR exception is raised.

The second routine, Save_Node, places a node in the available space list.

The third routine, Save_List, places a list of nodes in the available space list.

The decomposition for this part is the same as that shown in the Top-Level Design Document.

3.3.7.1.9.1.1 REQUIREMENTS ALLOCATION

This part helps meet CAMP requirements R164, R165, R167.

3.3.7.1.9.1.2 LOCAL ENTITIES DESIGN

None.

3.3.7.1.9.1.3 INPUT/OUTPUT

GENERIC PARAMETERS:

Data types:

The following table summarizes the generic formal types required by this part:

Name	Type	Description
Nodes	limited private	A single element in the available space list
Pointers	access Nodes	A pointer to an element in the available space list

Data objects:

The following table summarizes the generic formal objects required by this part. All of these objects are in/out parameters and are changed by calles to the enclosed routines.

Name	Type	Value	Description
Available_ Length	INTEGER	N/A	Length of the available space list
Available_Head	Pointers	N/A	Points to the first element in the available space list
Available_Tail	Pointers	N/A	Points to the last element in the available space list

Subprograms:

The following table describes the generic formal subprograms required by this part:

Name	Type	Description
Dot_Next	function	Given a pointer to a node, this function returns a pointer to the next node in the list
Set_Next	procedure	Given two points, A and B, sets A.Next equal to B

3.3.7.1.9.1.4 LOCAL DATA

None.

3.3.7.1.9.1.5 PROCESS CONTROL

Not applicable.

3.3.7.1.9.1.6 PROCESSING

The following describes the processing performed by this part:

```
generic
   type Nodes              is limited private;
   type Pointers           is access Nodes;
   Available_Length        : in out INTEGER;
   Available_Head          : in out Pointers;
   Available_Tail          : in out Pointers;
   with function  Dot_Next (Ptr : in Pointers) return Pointers is <>;
   with procedure Set_Next (Ptr          : in Pointers;
                            Ptr_dot_Next : in Pointers) is <>;
package Available_Space_List_Operations is

   function  New_Node return Pointers;

   procedure Save_Node (Saved_Node : in Pointers);

   procedure Save_List (Saved_Head : in Pointers;
                        Saved_Tail : in Pointers;
                        Node_Count : in POSITIVE);

end Available_Space_List_Operations;
```

3.3.7.1.9.1.7 UTILIZATION OF OTHER ELEMENTS

None.

3.3.7.1.9.1.8 LIMITATIONS

The following table describes the exceptions raised by this part:

Name	When/Why Raised
STANDARD.STORAGE_ERROR	Raised during elaboration of this package if an attempt is made to allocate memory when no more is available

3.3.7.1.9.1.9 LLCSC DESIGN

None.

3.3.7.1.9.1.10 UNIT DESIGN

None.

3.3.7.1.9.2 AVAILABLE_SPACE_LIST_OPERATIONS PACKAGE DESIGN

This package contains a set of routines used to manipulate an available space list which is maintained local to the part instantiating this package.

The first routine, New_Node, will return a node to the calling routine. If a node is available in the available space list, the node will be retrieved from there. If not, a new node will be dynamically allocated. If no memory is available for the allocation, a STORAGE_ERROR exception is raised.

The second routine, Save_Node, places a node in the available space list.

The third routine, Save_List, places a list of nodes in the available space list.

The decomposition for this part is the same as that shown in the Top-Level Design Document.

3.3.7.1.9.2.1 REQUIREMENTS ALLOCATION

This part helps meet CAMP requirements R164, R165, R167.

3.3.7.1.9.2.2 LOCAL ENTITIES DESIGN

None.

3.3.7.1.9.2.3 INPUT/OUTPUT

GENERIC PARAMETERS:

The following generic parameters were previously defined when this was specified in the package body of Abstract_Data_Structures.

Data types:

The following table summarizes the generic formal types required by this part:

Name	Type	Description
Nodes	limited private	A single element in the available space list
Pointers	access Nodes	A pointer to an element in the available space list

Data objects:

The following table summarizes the generic formal objects required by this part. All of these objects are in/out parameters and are changed by calls to the enclosed routines.

Name	Type	Value	Description
Available_Length	INTEGER	N/A	Length of the available space list
Available_Head	Pointers	N/A	Points to the first element in the available space list
Available_Tail	Pointers	N/A	Points to the last element in the available space list

Subprograms:

The following table describes the generic formal subprograms required by this part:

Name	Type	Description
Get_Next	function	Given a pointer to a node, this function returns a pointer to the next node in the list
Set_Next	procedure	Given two points, A and B, sets A.Next equal to B

3.3.7.1.9.2.4 LOCAL DATA

None.

3.3.7.1.9.2.5 PROCESS CONTROL

Not applicable.

3.3.7.1.9.2.6 PROCESSING

The following describes the processing performed by this part:

```
separate (Abstract_Data_Structures)
package body Available_Space_List_Operations is

end Available_Space_List_Operations;
```

3.3.7.1.9.2.7 UTILIZATION OF OTHER ELEMENTS

None.

3.3.7.1.9.2.8 LIMITATIONS

None.

3.3.7.1.9.2.9 LLCSC DESIGN

None.

3.3.7.1.9.2.10 UNIT DESIGN

3.3.7.1.9.2.10.1 NEW_NODE UNIT DESIGN

This function returns a node to the calling routine. If nodes are available in the space list, the node returned will be from there. If the available space list is empty, this routine will attempt to dynamically allocate memory. If no more memory is available on the system, a STORAGE_ERROR exception will be raised.

3.3.7.1.9.2.10.1.1 REQUIREMENTS ALLOCATION

This part helps meets CAMP requirements R164, R164, R176.

3.3.7.1.9.2.10.1.2 LOCAL ENTITIES DESIGN

None.

3.3.7.1.9.2.10.1.3 INPUT/OUTPUT

None.

3.3.7.1.9.2.10.1.4 LOCAL DATA

Data objects:

The following table describes the data objects maintained by this part:

Name	Type	Value	Description
Ptr	Pointers	N/A	Points to the node being returned
New_Available_Head	Pointers	N/A	Temporary variable used to mark where Available_Head will point when this routine is exited

3.3.7.1.9.2.10.1.5 PROCESS CONTROL

Not applicable.

3.3.7.1.9.2.10.1.6 PROCESSING

The following describes the processing performed by this part:

```
   function New_Node return Pointers is

--    ---------------------
--    --declaration section
--    ---------------------

      Ptr                : Pointers;
      New_Available_Head : Pointers;

-- -------------------------
-- --begin function New_Node
-- -------------------------

   begin

      if Available_Length > 0 then

--       --get the node from the available space list and mark the node
--       --that will now be the head of the available space list
         Ptr                := Available_Head;
         New_Available_Head := Dot_Next(Available_Head);

--       --initialize node being returned
         Set_Next (Ptr          => Ptr,
                   Ptr_dot_Next => NULL);

--       --adjust the available space list
         Available_Head   := New_Available_Head;
         Available_Length := Available_Length - 1;

      else

--       --allocate space to get the node
         Ptr := NEW Nodes;

      end if;
```

```
      return Ptr;

   end New_Node;
```

3.3.7.1.9.2.10.1.7 UTILIZATION OF OTHER ELEMENTS

UTILIZATION OF OTHER ELEMENTS IN TOP LEVEL COMPONENT:

The following tables describe the elements used by this part but defined elsewhere in the parent top level component:

Subprograms and task entries:

The following table describes the subroutines required by this part and defined as generic formal subprograms to the Available_Space_List_Operations package:

Name	Type	Description
Dot_Next	function	Given a pointer to a node, this function returns a pointer to the next node in the list
Set_Next	procedure	Given two points, A and B, sets A.Next equal to B

Data types:

The following table summarizes the types required by this part and defined as generic formal parameters to the Abstract_Data_Structures. Available_Space_-List_Operations package.

Name	Type	Description
Nodes	limited private	A single element in the available space list
Pointers	access Nodes	A pointer to an element in the available space list

Data objects:

The following table summarizes the objects required by this part and defined as generic formal parameters to the Abstract_Data_Structures. Available_Space_-List_Operations package.

Name	Type	Value	Description
Available_Length	INTEGER	N/A	Length of the available space list
Available_Head	Pointers	N/A	Points to the first element in the available space list

3.3.7.1.9.2.10.1.8 LIMITATIONS

The following table describes the exceptions raised by this part:

Name	When/Why Raised
STANDARD.STORAGE_ERROR	Raised if an attempt is made to allocate memory when no more is available

3.3.7.1.9.2.10.2 SAVE_NODE UNIT DESIGN

This procedure returns a node to the available space list. The node returned to the list is the one pointed to by Saved_Node.

3.3.7.1.9.2.10.2.1 REQUIREMENTS ALLOCATION

This part helps meets CAMP requirements R164, R164, R176.

3.3.7.1.9.2.10.2.2 LOCAL ENTITIES DESIGN

None.

3.3.7.1.9.2.10.2.3 INPUT/OUTPUT

FORMAL PARAMETERS:

The following table describes this part's formal parameters:

Name	Type	Mode	Description
Saved_Node	Pointers	in	Pointer to the node which is to be placed in the available space list

3.3.7.1.9.2.10.2.4 LOCAL DATA

None.

3.3.7.1.9.2.10.2.5 PROCESS CONTROL

Not applicable.

3.3.7.1.9.2.10.2.6 PROCESSING

The following describes the processing performed by this part:

```
procedure Save_Node(Saved_Node : in Pointers) is
```

```
   begin

      Set_Next (Ptr          => Available_Tail,
                Ptr_dot_Next => Saved_Node);

      Available_Tail := Saved_Node;

      Set_Next (Ptr          => Available_Tail,
                Ptr_dot_Next => NULL);

      Available_Length := Available_Length + 1;

   end Save_Node;
```

3.3.7.1.9.2.10.2.7 UTILIZATION OF OTHER ELEMENTS

UTILIZATION OF OTHER ELEMENTS IN TOP LEVEL COMPONENT:

The following tables describe the elements used by this part but defined elsewhere in the parent top level component:

Subprograms and task entries:

The following table describes the subroutines required by this part and defined as generic formal subprograms to the Available_Space_List_Operations package:

Name	Type	Description
Set_Next	procedure	Given two points, A and B, sets A.Next equal to B

Data types:

The following table summarizes the types required by this part and defined as generic formal parameters to the Abstract_Data_Structures. Available_Space_-List_Operations package.

Name	Type	Description
Nodes	limited private	A single element in the available space list
Pointers	access Nodes	A pointer to an element in the available space list

Data objects:

The following table summarizes the objects required by this part and defined as generic formal parameters to the Abstract_Data_Structures. Available_Space_-List_Operations package.

Name	Type	Value	Description
Available_ Length	INTEGER	N/A	Length of the available space list
Available_Tail	Pointers	N/A	Points to the last element in the available space list

3.3.7.1.9.2.10.2.8 LIMITATIONS

None.

3.3.7.1.9.2.10.3 SAVE_LIST UNIT DESIGN

This procedures places a linked list of nodes in the available space list.

3.3.7.1.9.2.10.3.1 REQUIREMENTS ALLOCATION

None.

3.3.7.1.9.2.10.3.2 LOCAL ENTITIES DESIGN

None.

3.3.7.1.9.2.10.3.3 INPUT/OUTPUT

FORMAL PARAMETERS:

The following table describes this part's formal parameters:

Name	Type	Mode	Description
Saved_Head	Pointers	in	Pointer to the first node to be placed in the available space list
Saved_Tail	Pointers	in	Pointer to the last node to be placed in the available space list
Node_Count	POSITIVE	in	Number of nodes to be placed in the available space list

3.3.7.1.9.2.10.3.4 LOCAL DATA

None.

3.3.7.1.9.2.10.3.5 PROCESS CONTROL

Not applicable.

3.3.7.1.9.2.10.3.6 PROCESSING

The following describes the processing performed by this part:

```
procedure Save_List (Saved_Head : in Pointers;
                     Saved_Tail : in Pointers;
                     Node_Count : in POSITIVE) is

begin

   Set_Next (Ptr           => Available_Tail,
             Ptr_dot_Next  => Saved_Head);

   Available_Tail := Saved_Tail;

   Set_Next (Ptr           => Available_Tail,
             Ptr_dot_Next  => Saved_Head);

   Available_Length := Available_Length + Node_Count;

end Save_List;
```

3.3.7.1.9.2.10.3.7 UTILIZATION OF OTHER ELEMENTS

UTILIZATION OF OTHER ELEMENTS IN TOP LEVEL COMPONENT:

The following tables describe the elements used by this part but defined elsewhere in the parent top level component:

Subprograms and task entries:

The following table describes the subroutines required by this part and defined as generic formal subprograms to the Available_Space_List_Operations package:

Name	Type	Description
Set_Next	procedure	Given two points, A and B, sets A.Next equal to B

Data types:

The following table summarizes the types required by this part and defined as generic formal parameters to the Available_Space_List_Operations LLCSC:

Name	Type	Description
Pointers	access Nodes	A pointer to an element in the available space list

Data objects:

The following table summarizes the objects required by this part and defined as generic formal parameters to the Available_Space_List_Operations LLCSC:

Name	Type	Value	Description
Available_ Length	INTEGER	N/A	Length of the available space list
Available_Tail	Pointers	N/A	Points to the last element in the available space list

3.3.7.1.9.2.10.3.8 LIMITATIONS

None.

3.3.7.1.9.3 BOUNDED_FIFO_BUFFER PACKAGE DESIGN (CATALOG #P331-0)

This generic package defines the data type and contains the operations required to perform first-in-first-out buffering operations on incoming data. The head always points to a dummy node. The first node following the dummy node contains the next piece of data to be retrieved. The tail always points to where the next element should be added. If the tail points to the element immediately in front of the head, the buffer is empty. If the tail points to the same element as the head, the buffer is full. Since the buffer is implemented as an array, the head and tail will advance through the array in a circular fashion, but no overwriting of data currently in the buffer will be permitted.

Empty FIFO buffer: +-+ <------Head +-+ +-+ +-+ <-----Tail +-+ +-+ +-+ +-+

Full FIFO buffer: Tail----->+-+ <------Head +-+ +-+ +-+ +-+ +-+ +-+ +-+

The decomposition for this part is the same as that shown in the Top-Level Design Document.

3.3.7.1.9.3.1 REQUIREMENTS ALLOCATION

This part meets CAMP required R125.

3.3.7.1.9.3.2 LOCAL ENTITIES DESIGN

None.

3.3.7.1.9.3.3 INPUT/OUTPUT

GENERIC PARAMETERS:

The following generic parameters were previously defined when this part was specified in the package specification of the Abstract_Data_Structures package:

Data types:

The following table summarizes the generic formal types required by this part:

Name	Type	Description
Elements	private	User defined type of data contained in the buffer

Data objects:

The following table summarizes the generic formal objects required by this part:

Name	Type	Value	Description
Initial_Buffer_Size	POSITIVE	N/A	Maximum number of elements which can be in the buffer at any given time

3.3.7.1.9.3.4 LOCAL DATA

None.

3.3.7.1.9.3.5 PROCESS CONTROL

Not applicable.

3.3.7.1.9.3.6 PROCESSING

The following describes the processing performed by this part:

```
separate (Abstract_Data_Structures)
package body Bounded_FIFO_Buffer is

end Bounded_FIFO_Buffer;
```

3.3.7.1.9.3.7 UTILIZATION OF OTHER ELEMENTS

UTILIZATION OF OTHER ELEMENTS IN TOP LEVEL COMPONENT:

The following tables describe the elements used by this part but defined elsewhere in the parent top level component:

Data types:

The following table describes the data types which were previously defined in this part's specification:

Name	Type	Range	Description
Buffer_Range	NATURAL subtype	0 .. Buffer_Size	Used to dimension the list of elements
Buffer_Statuses	discrete type	Empty, Available, Full	Used to indicate the status of the buffer

The following table describes the data types defined in the private part of this part's specification:

Name	Type	Range	Description
Buffers	record	N/A	List of data along with relevant information
Lists	array	N/A	Array of elements

Data objects:

The following table describes the data objects which were previously defined in this part's specification:

Name	Type	Value	Description
Buffer_Size	POSITIVE	Initial_Buffer_Size	Number of usable elements in a buffer

Exceptions:

The following table describes the exceptions which were previously defined in this part's specification:

Name	Description
Buffer_Empty	Error condition raised if an attempt is made to look at or retrieve elements from an empty buffer
Buffer_Full	Error condition raised if an attempt is made to add elements to a full buffer

3.3.7.1.9.3.8 LIMITATIONS

None.

3.3.7.1.9.3.9 LLCSC DESIGN

None.

3.3.7.1.9.3.10 UNIT DESIGN

3.3.7.1.9.3.10.1 CLEAR_BUFFER UNIT DESIGN

This procedure clears an input buffer by setting its length to 0 and resetting its head and tail to 0 and 1, respectively.

3.3.7.1.9.3.10.1.1 REQUIREMENTS ALLOCATION

This part meets CAMP requirement R125.

3.3.7.1.9.3.10.1.2 LOCAL ENTITIES DESIGN

None.

3.3.7.1.9.3.10.1.3 INPUT/OUTPUT

FORMAL PARAMETERS:

The following table describes this part's formal parameters:

Name	Type	Mode	Description
Buffer	Buffers	out	FIFO buffer being accessed

3.3.7.1.9.3.10.1.4 LOCAL DATA

None.

3.3.7.1.9.3.10.1.5 PROCESS CONTROL

Not applicable.

3.3.7.1.9.3.10.1.6 PROCESSING

The following describes the processing performed by this part:

```
   procedure Clear_Buffer (Buffer : out Buffers) is

--    ---------------------
--    --declaration section
--    ---------------------

      Buffer_Length : Buffer_Range renames Buffer.Buffer_Length;
```

```
      Head          : Buffer_Range renames Buffer.Head;
      Tail          : Buffer_Range renames Buffer.Tail;

-- ------------------------------
-- --begin procedure Clear_Buffer
-- ------------------------------

   begin

      Buffer_Length := 0;
      Head          := 0;
      Tail          := 1;

   end Clear_Buffer ;
```

3.3.7.1.9.3.10.1.7 UTILIZATION OF OTHER ELEMENTS

None.

3.3.7.1.9.3.10.1.8 LIMITATIONS

None.

3.3.7.1.9.3.10.2 ADD_ELEMENT UNIT DESIGN

This procedure adds an element to an input buffer if the buffer is not already full. After the element is added, the tail is advanced one place in the buffer and the length counter is incremented by 1.

The exception Buffer_Full is raised if an attempt is made to add an element to an already full buffer.

3.3.7.1.9.3.10.2.1 REQUIREMENTS ALLOCATION

This part meets CAMP requirement R125.

3.3.7.1.9.3.10.2.2 LOCAL ENTITIES DESIGN

None.

3.3.7.1.9.3.10.2.3 INPUT/OUTPUT

FORMAL PARAMETERS:

The following table describes this part's formal parameters:

Name	Type	Mode	Description
Buffer	Buffers	in out	FIFO buffer being accessed
New_Element	Elements	in	Element to be added to the buffer

3.3.7.1.9.3.10.2.4 LOCAL DATA

None.

3.3.7.1.9.3.10.2.5 PROCESS CONTROL

Not applicable.

3.3.7.1.9.3.10.2.6 PROCESSING

The following describes the processing performed by this part:

```
   procedure Add_Element (New_Element : in     Elements;
                          Buffer      : in out Buffers) is

--    ---------------------
--    --declaration section
--    ---------------------

      List          : Lists        renames Buffer.List;
      Buffer_Length : Buffer_Range renames Buffer.Buffer_Length;
      Head          : Buffer_Range renames Buffer.Head;
      Tail          : Buffer_Range renames Buffer.Tail;

-- ----------------------------
-- --begin procedure Add_Element
-- ----------------------------

   begin

--    --make sure buffer isn't full
      if Head = Tail then
         raise Buffer_Full;
      end if;

      List(Tail) := New_Element;
      Buffer_Length := Buffer_Length + 1;
      if Tail = Buffer_Size then
         Tail := 0;
      else
         Tail := Tail + 1;
      end if;

   end Add_Element ;
```

3.3.7.1.9.3.10.2.7 UTILIZATION OF OTHER ELEMENTS

UTILIZATION OF OTHER ELEMENTS IN TOP LEVEL COMPONENT:

The following tables describe the elements used by this part but defined elsewhere in the parent top level component:

Data types:

The following table summarizes the types required by this part and defined as generic formal types to the Abstract_Data_Structures. Bounded_FIFO_Buffer package:

Name	Type	Description
Elements	private	User defined type of data contained in the buffer

The following table summarizes the types required by this part and defined in the package specification of Abstract_Data_Structures. Bounded_FIFO_Buffer.

Name	Type	Range	Description
Buffer_Range	NATURAL subtype	0 .. Buffer_Size	Used to dimension the list of elements

The following table describes the data types defined in the private part of the Abstract_Data_Structures.Bounded_FIFO_Buffer package:

Name	Type	Range	Description
Buffers	record	N/A	List of data along with relevant information
Lists	array	N/A	Array of elements

Data objects:

The following table summarizes the objects required by this part and defined in the package specification of Abstract_Data_Structures. Bounded_FIFO_Buffer:

Name	Type	Value	Description
Buffer_Size	POSITIVE	Initial_ Buffer_Size	Number of usable elements in a buffer

Exceptions:

The following table summarizes the exceptions required by this part and defined in the package specification of Abstract_Data_Structures. Bounded_FIFO_Buffer:

Name	Description
Buffer_Full	Error condition raised if an attempt is made to add elements to a full buffer

3.3.7.1.9.3.10.2.8 LIMITATIONS

The following table describes the exceptions raised by this part:

Name	Description
Buffer_Full	Error condition raised if an attempt is made to add elements to a full buffer

3.3.7.1.9.3.10.3 RETRIEVE_ELEMENT UNIT DESIGN

This procedure retrieves the top element in the buffer if the buffer is not empty. The head is advanced through the buffer by 1 before the element is retrieved and the size of the buffer is decremented by 1 after the element is retrieved.

If the buffer is empty before calling this routine, the exception Buffer_Empty is raised.

3.3.7.1.9.3.10.3.1 REQUIREMENTS ALLOCATION

This part meets CAMP requirement R125.

3.3.7.1.9.3.10.3.2 LOCAL ENTITIES DESIGN

None.

3.3.7.1.9.3.10.3.3 INPUT/OUTPUT

FORMAL PARAMETERS:

The following table describes this part's formal parameters:

Name	Type	Mode	Description
Buffer	Buffers	in out	FIFO buffer being accessed
Old_Element	Elements	out	Element retrieved from the buffer

3.3.7.1.9.3.10.3.4 LOCAL DATA

None.

3.3.7.1.9.3.10.3.5 PROCESS CONTROL

Not applicable.

3.3.7.1.9.3.10.3.6 PROCESSING

The following describes the processing performed by this part:

```
   procedure Retrieve_Element (Buffer      : in out Buffers;
                               Old_Element :    out Elements) is

--     --------------------
--     --declaration section
--     --------------------

       Buffer_Length : Buffer_Range renames Buffer.Buffer_Length;
       Head          : Buffer_Range renames Buffer.Head;
       List          : Lists        renames Buffer.List;
       Tail          : Buffer_Range renames Buffer.Tail;

-- ---------------------------------
-- --begin procedure Retrieve_Element
-- ---------------------------------

   begin

--     --make sure don't have an empty buffer
       if Head = (Tail-1) or else (Tail = 0 and Head = Buffer_Size) then
          raise Buffer_Empty;
       end if;

       if Head = Buffer_Size then
          Head := 0;
       else
          Head := Head + 1;
       end if;
       Old_Element   := List(Head);
       Buffer_Length := Buffer_Length - 1;

   end Retrieve_Element ;
```

3.3.7.1.9.3.10.3.7 UTILIZATION OF OTHER ELEMENTS

UTILIZATION OF OTHER ELEMENTS IN TOP LEVEL COMPONENT:

The following tables describe the elements used by this part but defined elsewhere in the parent top level component:

Data types:

The following table summarizes the types required by this part and defined as generic formal types to the Abstract_Data_Structures. Bounded_FIFO_Buffer package:

Name	Type	Description
Elements	private	User defined type of data contained in the buffer

The following table summarizes the types required by this part and defined in the package specification of Abstract_Data_Structures. Bounded_FIFO_Buffer.

Name	Type	Range	Description
Buffer_Range	NATURAL subtype	0 .. Buffer_Size	Used to dimension the list of elements

The following table describes the data types defined in the private part of the Abstract_Data_Structures.Bounded_FIFO_Buffer package:

Name	Type	Range	Description
Buffers	record	N/A	List of data along with relevant information
Lists	array	N/A	Array of elements

Exceptions:

The following table summarizes the exceptions required by this part and defined in the package specification of Abstract_Data_Structures. Bounded_FIFO_Buffer:

Name	Description
Buffer_Empty	Error condition raised if an attempt is made to look at or retrieve elements from an empty buffer

3.3.7.1.9.3.10.3.8 LIMITATIONS

The following table describes the exceptions raised by this part:

Name	Description
Buffer_Empty	Error condition raised if an attempt is made to look at or retrieve elements from an empty buffer

3.3.7.1.9.3.10.4 PEEK UNIT DESIGN

This function returns the first element of the buffer if the buffer is not empty. The status of the buffer is not changed, however, and the element itself remains in the buffer.

The Buffer_Empty exception is raised if an attempt is made to look at an empty buffer.

3.3.7.1.9.3.10.4.1 REQUIREMENTS ALLOCATION

This part meets CAMP requirement R125.

3.3.7.1.9.3.10.4.2 LOCAL ENTITIES DESIGN

None.

3.3.7.1.9.3.10.4.3 INPUT/OUTPUT

FORMAL PARAMETERS:

The following table describes this part's formal parameters:

Name	Type	Mode	Description
Buffer	Buffers	in out	FIFO buffer being accessed

3.3.7.1.9.3.10.4.4 LOCAL DATA

Data objects:

The following table describes the data objects maintained local to this part:

Name	Type	Description
Spot	Buffer_Range	Marks location of element to be looked at

3.3.7.1.9.3.10.4.5 PROCESS CONTROL

Not applicable.

3.3.7.1.9.3.10.4.6 PROCESSING

The following describes the processing performed by this part:

```
function Peek (Buffer : in Buffers) return Elements is
```

```
--    ---------------------
--    --declaration section
--    ---------------------

      Buffer_Length : Buffer_Range renames Buffer.Buffer_Length;
      Head          : Buffer_Range renames Buffer.Head;
      Tail          : Buffer_Range renames Buffer.Tail;
      List          : Lists        renames Buffer.List;
      Spot          : Buffer_Range;

-- ---------------------
-- --begin function Peek
-- ---------------------

   begin

--    --make sure don't have an empty buffer
      if Head = (Tail-1) or else (Tail = 0 and Head = Buffer_Size) then
         raise Buffer_Empty;
      end if;

      if Head = Buffer_Size then
         Spot := 0;
      else
         Spot := Head + 1;
      end if;

      return List(Spot);

   end Peek ;
```

3.3.7.1.9.3.10.4.7 UTILIZATION OF OTHER ELEMENTS

UTILIZATION OF OTHER ELEMENTS IN TOP LEVEL COMPONENT:

The following tables describe the elements used by this part but defined elsewhere in the parent top level component:

Data types:

The following table summarizes the types required by this part and defined as generic formal types to the Abstract_Data_Structures. Bounded_FIFO_Buffer package:

Name	Type	Description
Elements	private	User defined type of data contained in the buffer

The following table summarizes the types required by this part and defined in the package specification of Abstract_Data_Structures. Bounded_FIFO_Buffer.

Name	Type	Range	Description
Buffer_Range	NATURAL subtype	0 .. Buffer_Size	Used to dimension the list of elements

The following table describes the data types defined in the private part of the Abstract_Data_Structures.Bounded_FIFO_Buffer package:

Name	Type	Range	Description
Buffers	record	N/A	List of data along with relevant information
Lists	array	N/A	Array of elements

Exceptions:

The following table summarizes the exceptions required by this part and defined in the package specification of Abstract_Data_Structures. Bounded_FIFO_Buffer:

Name	Description
Buffer_Empty	Error condition raised if an attempt is made to look at or retrieve elements from an empty buffer

3.3.7.1.9.3.10.4.8 LIMITATIONS

The following table describes the exceptions raised by this part:

Name	Description
Buffer_Empty	Error condition raised if an attempt is made to look at or retrieve elements from an empty buffer

3.3.7.1.9.3.10.5 BUFFER_STATUS UNIT DESIGN

This function returns the status of the buffer. If there are no elements in the buffer, the status is empty; if there is no room for additional elements, the status is full; otherwise, the status is available.

3.3.7.1.9.3.10.5.1 REQUIREMENTS ALLOCATION

This part meets CAMP requirement R125.

3.3.7.1.9.3.10.5.2 LOCAL ENTITIES DESIGN

None.

3.3.7.1.9.3.10.5.3 INPUT/OUTPUT

FORMAL PARAMETERS:

The following table describes this part's formal parameters:

Name	Type	Mode	Description
Buffer	Buffers	in out	FIFO buffer being accessed

3.3.7.1.9.3.10.5.4 LOCAL DATA

Data objects:

The following objects are maintained local to this part:

Name	Type	Description
Status	Buffer_Statuses	Status of the buffer

3.3.7.1.9.3.10.5.5 PROCESS CONTROL

Not applicable.

3.3.7.1.9.3.10.5.6 PROCESSING

The following describes the processing performed by this part:

```
   function Buffer_Status (Buffer : in Buffers) return Buffer_Statuses is

--    ---------------------
--    --declaration section
--    ---------------------

      Head : Buffer_Range renames Buffer.Head;
      Tail : Buffer_Range renames Buffer.Tail;

      Status : Buffer_Statuses;

-- ------------------------------
-- --begin function Buffer_Status
-- ------------------------------

   begin
```

```
      if Head = (Tail-1) or else (Tail = 0 and Head = Buffer_Size) then
         Status := Empty;
      elsif Head = Tail then
         Status := Full;
      else
         Status := Available;
      end if;

      return Status;

   end Buffer_Status ;
```

3.3.7.1.9.3.10.5.7 UTILIZATION OF OTHER ELEMENTS

UTILIZATION OF OTHER ELEMENTS IN TOP LEVEL COMPONENT:

The following tables describe the elements used by this part but defined elsewhere in the parent top level component:

Data types:

The following table summarizes the types required by this part and defined in the package specification of Abstract_Data_Structures. Bounded_FIFO_Buffer.

Name	Type	Range	Description
Buffer_Range	NATURAL subtype	0 .. Buffer_Size	Used to dimension the list of elements
Buffer_Statuses	discrete type	Empty, Available, Full	Used to indicate the status of the buffer

The following table describes the data types defined in the private part of the Abstract_Data_Structures.Bounded_FIFO_Buffer package:

Name	Type	Range	Description
Buffers	record	N/A	List of data along with relevant information

Data objects:

The following table summarizes the objects required by this part and defined in the package specification of Abstract_Data_Structures. Bounded_FIFO_Buffer:

Name	Type	Value	Description
Buffer_Size	POSITIVE	Initial_Buffer_Size	Number of usable elements in a buffer

3.3.7.1.9.3.10.5.8 LIMITATIONS

None.

3.3.7.1.9.3.10.6 BUFFER_LENGTH UNIT DESIGN

This function returns the length of the current buffer.

3.3.7.1.9.3.10.6.1 REQUIREMENTS ALLOCATION

This part meets CAMP requirement R125.

3.3.7.1.9.3.10.6.2 LOCAL ENTITIES DESIGN

None.

3.3.7.1.9.3.10.6.3 INPUT/OUTPUT

FORMAL PARAMETERS:

The following table describes this part's formal parameters:

Name	Type	Mode	Description
Buffer	Buffers	in out	FIFO buffer being accessed

3.3.7.1.9.3.10.6.4 LOCAL DATA

None.

3.3.7.1.9.3.10.6.5 PROCESS CONTROL

Not applicable.

3.3.7.1.9.3.10.6.6 PROCESSING

The following describes the processing performed by this part:

```
function Buffer_Length (Buffer : in Buffers) return Buffer_Range is
begin
   return Buffer.Buffer_Length;
end Buffer_Length ;
```

3.3.7.1.9.3.10.6.7 UTILIZATION OF OTHER ELEMENTS

UTILIZATION OF OTHER ELEMENTS IN TOP LEVEL COMPONENT:

The following tables describe the elements used by this part but defined elsewhere in the parent top level component:

Data types:

The following table summarizes the types required by this part and defined in the package specification of Abstract_Data_Structures. Bounded_FIFO_Buffer.

Name	Type	Range	Description
Buffer_Range	NATURAL subtype	0 .. Buffer_Size	Used to dimension the list of elements

The following table describes the data types defined in the private part of the Abstract_Data_Structures.Bounded_FIFO_Buffer package:

Name	Type	Range	Description
Buffers	record	N/A	List of data along with relevant information

3.3.7.1.9.3.10.6.8 LIMITATIONS

None.

3.3.7.1.9.4 UNBOUNDED_FIFO_BUFFER PACKAGE DESIGN (CATALOG #P332-0)

This generic package defines the data type and contains the operations required to perform first-in-first-out buffering operations on incoming data. The head of the buffer always points to a dummy node. The first node following the dummy node contains the next piece of data to be retrieved. The tail always points to the node containing the last element added to the buffer. If the tail points to the same node as the head, the buffer is empty.

A buffer must be initialized before it is used. If an attempt is made to use an uninitialized buffer, the exception Buffer_Not_Initialized will be raised. The Initialized_Buffer procedure returns an initialized buffer. The Clear_Buffer procedure returns the nodes of a buffer to the available space list and then returns an initialized buffer.

An available space list is maintained local to this part. When this part is elaborated the available space list will have a dummy node plus Initial_Available_Space_Size nodes. When nodes are added to the buffer, the Add_Element routine will try to get a node from the available space list before attempting to allocate more memory. When the Retrieve_Element routine is called, the unused node will be returned to the available space list for later use. The memory committed to the available space may be deallocated by calling

the Free_Memory procedure.

The decomposition for this part is the same as that shown in the Top-Level Design Document.

3.3.7.1.9.4.1 REQUIREMENTS ALLOCATION

This part meets CAMP requirement R164.

3.3.7.1.9.4.2 LOCAL ENTITIES DESIGN

Data structures:

An available space list is maintained local to this part's package body.

Subprograms:

The following subprograms are contained local to this body:

Name	Type	Description
Free_Node	procedure	Instantiation of UNCHECKED_DEALLOCATION
Dot_Next	function	Given a pointer P, this function returns the value of P.Next
Set_Next	procedure	Given two points P & Q, this procedure sets P.Next = Q

The following subprograms are contained in this part as a result of renaming operations on identically named routines contained in the locally instantiated Available_Space_Operations package.

Name	Type	Description
New_Node	function	Returns a node to the calling routine; will get a node from the available space list if possible, otherwise will allocate a new node
Save_Node	procedure	Handles placing a node in the available space list
Save_List	procedure	Handles placing a list of nodes in the available space list

This package body contains code to initialize the Available_Space_List. This code is executed when the package is elaborated. At a minimum, this code calls the Initialize_Buffer procedure to initialize the Available_Space_List so it contains a dummy node. If the generic formal object Initial_Available_Space_Size is greater than or equal to 1, this routine then places the requested number of nodes (in addition to the dummy node) in the available space list.

3.3.7.1.9.4.3 INPUT/OUTPUT

GENERIC PARAMETERS:

The following generic parameters were previously defined when this part was specified in the package specification of the Abstract_Data_Structures package:

Data types:

The following table summarizes the generic formal types required by this part:

Name	Type	Description
Elements	private	User defined type of data contained in the buffer

Data objects:

The following table summarizes the generic formal objects required by this part:

Name	Type	Description
Initial_Available_Space_Size	NATURAL	Number of nodes to be initially placed in the available space list

3.3.7.1.9.4.4 LOCAL DATA

None.

3.3.7.1.9.4.5 PROCESS CONTROL

Not applicable.

3.3.7.1.9.4.6 PROCESSING

The following describes the processing performed by this part:

```
with UNCHECKED_DEALLOCATION;
separate (Abstract_Data_Structures)
package body Unbounded_FIFO_Buffer is

-- ---------------------
-- --declaration section
-- ---------------------

-- --this variable is accessed ONLY when setting up the available space list
   Initial_Head    : Pointers := new Nodes;

   Available_Space : Buffers := (Current_Length => 0,
                                 Head           => Initial_Head,
```

```
                                    Tail           => Initial_Head);

    Available_Length : INTEGER  renames Available_Space.Current_Length;
    Available_Head   : Pointers renames Available_Space.Head;
    Available_Tail   : Pointers renames Available_Space.Tail;

    procedure Free is new UNCHECKED_DEALLOCATION
                          (Object => Nodes,
                           Name   => Pointers);

    procedure Free_Node (Which_Node : in out Pointers)
              renames Free;

    function Dot_Next (Ptr : in Pointers) return Pointers;

    procedure Set_Next (Ptr          : in Pointers;
                        Ptr_dot_Next : in Pointers);

    package Available_Space_Operations is new
            Available_Space_List_Operations
               (Nodes                    => Nodes,
                Pointers                 => Pointers,
                Available_Length         => Available_Length,
                Available_Head           => Available_Head,
                Available_Tail           => Available_Tail);

    function New_Node return Pointers
             renames Available_Space_Operations.New_Node;

    procedure Save_Node (Saved_Node : in Pointers)
             renames Available_Space_Operations.Save_Node;

    procedure Save_List (Saved_Head : in Pointers;
                         Saved_Tail : in Pointers;
                         Node_Count : in POSITIVE)
             renames Available_Space_Operations.Save_List;


-----------------------------------------
--begin package Unbounded_FIFO_Buffer
--(see header for package body for details)
-----------------------------------------
begin

-- --set up available space list if one is desired
   if Initial_Available_Space_Size > 0 then

      Add_Nodes_To_Available_Space_List:
         for I in 1..Initial_Available_Space_Size loop
            Available_Tail.Next := NEW Nodes;
            Available_Tail      := Available_Tail.Next;
         end loop Add_Nodes_to_Available_Space_List;

      Available_Length := Initial_Available_Space_Size;

   end if;
```

```
end Unbounded_FIFO_Buffer;
```

3.3.7.1.9.4.7 UTILIZATION OF OTHER ELEMENTS

The following library units are with'd by this part:

1. Unchecked_Deallocation

Subprograms and task entries:

The following table describes the subroutines required by this part:

Name	Type	Source	Description
UNCHECKED_DEALLOCATION	generic function	N/A	Used to deallocate memory

Exceptions:

The following table describes the exceptions required by this part and defined in the Ada predefined package STANDARD:

Name	Description
STORAGE_ERROR	Raised when an attempt is made to dynamically allocate more memory than is available

UTILIZATION OF OTHER ELEMENTS IN TOP LEVEL COMPONENT:

The following tables describe the elements used by this part but defined elsewhere in the parent top level component:

Packages:

The following table describes the packages required by this part and specified in the package body of the Abstract_Data_Structures package:

Name	Type	Description
Available_Space_List_Operations	generic package	Contains the routines required to retrieve a node from and place a node in the available space list

Data types:

The following data types were previously defined in this part's package specification:

Name	Type	Range	Description
Buffer_Statuses	discrete type	Empty, Available, Uninitialized	Used to indicate the status of the buffer

The following data types were previously defined in the private portion of this part's package specification:

Name	Type	Range	Description
Nodes	record	N/A	A single entity in the buffer; contains data and a pointer to the next node
Pointers	access	N/A	Points to a node in the buffer
Buffers	record	N/A	Record containing the value of the current length, head, and tail of the buffer

Exceptions:

The following exceptions were previously defined in this part's package specification:

Name	Description
Buffer_Empty	Error condition raised if an attempt is made to look at or retrieve elements from an empty buffer
Buffer_Not_Initialized	Raised if an attempt is made to use an uninitialized buffer

3.3.7.1.9.4.8 LIMITATIONS

The following table describes the exceptions raised by this part:

Name	When/Why Raised
Storage_Error	Raised during elaboration of this package if an attempt is made to allocate memory when no more is available

3.3.7.1.9.4.9 LLCSC DESIGN

None.

3.3.7.1.9.4.10 UNIT DESIGN

3.3.7.1.9.4.10.1 INITIALIZE_BUFFER UNIT DESIGN

This procedure initializes a buffer. It does this in the following manner:

1) If the buffer has never been initialized then: o places a dummy node in the buffer and o initializes the length to 0

2) else if the buffer has elements in it then: o calls the Clear_Buffer procedure

3) else if the buffer has a length of 0 then o does nothing

3.3.7.1.9.4.10.1.1 REQUIREMENTS ALLOCATION

This part meets CAMP requirement R164.

3.3.7.1.9.4.10.1.2 LOCAL ENTITIES DESIGN

None.

3.3.7.1.9.4.10.1.3 INPUT/OUTPUT

FORMAL PARAMETERS:

The following table describes this part's formal parameters:

Name	Type	Mode	Description
Buffer	Buffers	in out	FIFO buffer being initialized

3.3.7.1.9.4.10.1.4 LOCAL DATA

None.

3.3.7.1.9.4.10.1.5 PROCESS CONTROL

Not applicable.

3.3.7.1.9.4.10.1.6 PROCESSING

The following describes the processing performed by this part:

```
   procedure Initialize_Buffer (Buffer : in out Buffers) is

--    ---------------------
--    --declaration section
--    ---------------------
```

```
      Current_Length : INTEGER  renames Buffer.Current_Length;
      Head           : Pointers renames Buffer.Head;
      Tail           : Pointers renames Buffer.Tail;

-- ---------------------------------
-- --begin procedure Initialize_Buffer
-- ---------------------------------

   begin

      if Current_Length = -1 then

--       --handle an uninitialized buffer
         Head := New_Node;
         Tail := Head;
         Current_Length := 0;

      elsif Current_Length > 0 then

--       --handle a buffer that has something in it
         Clear_Buffer(Buffer => Buffer);

      else

--       --current length = 0 so it is already initialized
         NULL;

      end if;

   end Initialize_Buffer ;
```

3.3.7.1.9.4.10.1.7 UTILIZATION OF OTHER ELEMENTS

UTILIZATION OF OTHER ELEMENTS IN TOP LEVEL COMPONENT:

The following tables describe the elements used by this part but defined elsewhere in the parent top level component:

Subprograms and task entries:

The following table summarizes the subroutines and task entries required by this part and defined in the package specification of Unbounded_FIFO_Buffer:

Name	Type	Description
Clear_ Buffer	procedure	Returns all the nodes in a buffer to the available space list

Data types:

The following table summarizes the types required by this part and defined in the private portion of the part's package specification:

Name	Type	Range	Description
Nodes	record	N/A	A single entity in the buffer; contains data and a pointer to the next node
Pointers	access	N/A	Points to a node in the buffer
Buffers	record	N/A	Record containing the value of the current length, head, and tail of the buffer

3.3.7.1.9.4.10.1.8 LIMITATIONS

None.

3.3.7.1.9.4.10.2 CLEAR_BUFFER UNIT DESIGN

This procedure returns all the elements in a buffer, except for the dummy node, to the available space list. If this routine is sent an uninitialized buffer, a Buffer_Not_Initialized exception is raised.

3.3.7.1.9.4.10.2.1 REQUIREMENTS ALLOCATION

This part meets CAMP requirement R164.

3.3.7.1.9.4.10.2.2 LOCAL ENTITIES DESIGN

None.

3.3.7.1.9.4.10.2.3 INPUT/OUTPUT

FORMAL PARAMETERS:

The following table describes this part's formal parameters:

Name	Type	Mode	Description
Buffer	Buffers	in out	FIFO buffer being cleared

3.3.7.1.9.4.10.2.4 LOCAL DATA

Data objects:

The following table describes the objects maintained local to this part:

Name	Type	Description
This_Node	Pointers	Node to be placed in the available space list

3.3.7.1.9.4.10.2.5 PROCESS CONTROL

Not applicable.

3.3.7.1.9.4.10.2.6 PROCESSING

The following describes the processing performed by this part:

```
   procedure Clear_Buffer (Buffer : in out Buffers) is

--    ---------------------
--    --declaration section
--    ---------------------

      Current_Length : INTEGER  renames Buffer.Current_Length;
      Head           : Pointers renames Buffer.Head;
      Tail           : Pointers renames Buffer.Tail;

      This_Node      : Pointers;

-- ------------------------------
-- --begin procedure Clear_Buffer
-- ------------------------------

   begin

--    --make sure this is an initialized buffer
      if Current_Length = -1 then
         raise Buffer_Not_Initialized;
      end if;

--    --placed nodes in the available space list
      Save_List (Saved_Head => Head.Next,
                 Saved_Tail => Tail,
                 Node_Count => Current_Length);

--    --reinitialize buffer variables
      Current_Length := 0;
      Head.Next      := NULL;
      Tail           := Head;

   end Clear_Buffer ;
```

3.3.7.1.9.4.10.2.7 UTILIZATION OF OTHER ELEMENTS

UTILIZATION OF OTHER ELEMENTS IN TOP LEVEL COMPONENT:

The following tables describe the elements used by this part but defined elsewhere in the parent top level component:

Subprograms and task entries:

The following table summarizes the subroutines and task entries required by this part and defined in the package body of Unbounded_FIFO_Buffer:

Name	Type	Description
Save_List	procedure	Handles placing a list of nodes in the available space list

Data types:

The following table summarizes the types required by this part and defined in the private portion of the part's package specification:

Name	Type	Range	Description
Nodes	record	N/A	A single entity in the buffer; contains data and a pointer to the next node
Pointers	access	N/A	Points to a node in the buffer
Buffers	record	N/A	Record containing the value of the current length, head, and tail of the buffer

Exceptions:

The following table summarizes the exceptions required by this part and defined in the package specification of Abstract_Data_Structures. Unbounded_FIFO_-Buffer:

Name	Description
Buffer_Not_Initialized	Raised if an attempt is made to use an uninitialized buffer

3.3.7.1.9.4.10.2.8 LIMITATIONS

The following table describes the exceptions raised by this part:

Name	When/Why Raised
Buffer_Not_Initialized	Raised if an attempt is made to use an uninitialized buffer

3.3.7.1.9.4.10.3 FREE_MEMORY UNIT DESIGN

This procedure deallocates the memory occupied by the available space list.

3.3.7.1.9.4.10.3.1 REQUIREMENTS ALLOCATION

This part meets CAMP requirement R164.

3.3.7.1.9.4.10.3.2 LOCAL ENTITIES DESIGN

None.

3.3.7.1.9.4.10.3.3 INPUT/OUTPUT

None.

3.3.7.1.9.4.10.3.4 LOCAL DATA

Data objects:

The following table describes the data objects maintained by this part:

Name	Type	Value	Description
Node_to_be_Freed	Pointers	N/A	Pointer to the node to be deallocated

3.3.7.1.9.4.10.3.5 PROCESS CONTROL

Not applicable.

3.3.7.1.9.4.10.3.6 PROCESSING

The following describes the processing performed by this part:

```
   procedure Free_Memory is

--    ---------------------
--    --declaration section
--    ---------------------

      Node_to_be_Freed : Pointers;

-- ----------------------------
-- --begin procedure Free_Memory
-- ----------------------------

   begin

      Clear_Out_Available_Space_List:
         while Available_Head /= Available_Tail loop

            Node_To_Be_Freed := Available_Head;
            Available_Head   := Available_Head.Next;
```

```
            Free_Node (Which_Node => Node_to_be_Freed);

         end loop Clear_Out_Available_Space_List;

      Available_Length := 0;

   end Free_Memory ;
```

3.3.7.1.9.4.10.3.7 UTILIZATION OF OTHER ELEMENTS

UTILIZATION OF OTHER ELEMENTS IN TOP LEVEL COMPONENT:

The following tables describe the elements used by this part but defined elsewhere in the parent top level component:

Subprograms and task entries:

The following table summarizes the subroutines and task entries required by this part and defined in the package body of Unbounded_FIFO_Buffer:

Name	Type	Description
Free_Node	procedure	Instantiation of UNCHECKED_DEALLOCATION

Data types:

The following table summarizes the types required by this part and defined as generic parameters to the Abstract_Data_Structures. Unbounded_FIFO_Buffer package:

Name	Type	Description
Elements	private	User defined type of data contained in the buffer

The following table summarizes the types required by this part and defined in the private portion of the part's package specification:

Name	Type	Range	Description
Nodes	record	N/A	A single entity in the buffer; contains data and a pointer to the next node
Pointers	access	N/A	Points to a node in the buffer
Buffers	record	N/A	Record containing the value of the current length, head, and tail of the buffer

Data objects:

The following table summarizes the objects required by this part and defined in the package body of Unbounded_FIFO_Buffer:

Name	Type	Description
Available_ Space	Buffers	List of available nodes; nodes will be added to list when Retrieve_Element is called and retrieved from the list when Add_Element is called; the nodes in the list are deallocated when Clear_Memory is called

The following table summarizes the data objects required by this part and defined in the package body of Unbounded_FIFO_Buffer:

Name	Type	Value	Description
Available_ Length	INTEGER	Available_Space. Current_Length	Indicates the current length of the available space list
Available_ Head	Pointers	Available_Space. Head	Points to the head node in the available space list
Available_ Tail	Pointers	Available_Space. Tail	Points to the tail node in the available space list

3.3.7.1.9.4.10.3.8 LIMITATIONS

None.

3.3.7.1.9.4.10.4 ADD_ELEMENT UNIT DESIGN

This procedure adds an element to the end of the FIFO buffer.

If the buffer has not been initialized, the exception Buffer_Not_Initialized is raised.

The Storage_Error exception is raised if a call to this routine requires memory to be dynamically allocated when no more memory is available.

3.3.7.1.9.4.10.4.1 REQUIREMENTS ALLOCATION

This part meets CAMP requirement R164.

3.3.7.1.9.4.10.4.2 LOCAL ENTITIES DESIGN

None.

3.3.7.1.9.4.10.4.3 INPUT/OUTPUT

FORMAL PARAMETERS:

The following table describes this part's formal parameters:

Name	Type	Mode	Description
Buffer	Buffers	in out	FIFO buffer being accessed
New_Element	Elements	in	Element to be added to the buffer

3.3.7.1.9.4.10.4.4 LOCAL DATA

None.

3.3.7.1.9.4.10.4.5 PROCESS CONTROL

Not applicable.

3.3.7.1.9.4.10.4.6 PROCESSING

The following describes the processing performed by this part:

```
   procedure Add_Element (New_Element : in     Elements;
                          Buffer      : in out Buffers) is

--    ---------------------
--    --declaration section
--    ---------------------

      Current_Length : INTEGER  renames Buffer.Current_Length;
      Tail           : Pointers renames Buffer.Tail;

      New_Tail       : Pointers;

-- ----------------------------
-- --begin procedure Add_Element
-- ----------------------------

   begin

--    --make sure buffer has been initialized
      if Current_Length = -1 then
         raise Buffer_Not_Initialized;
      end if;

--    --now get a node
      New_Tail := New_Node;

--    --now adjust the buffer
      Tail.Next      := New_Tail;
      Tail           := New_Tail;
```

```
      Tail.Data      := New_Element;
      Current_Length := Current_Length + 1;

   end Add_Element ;
```

3.3.7.1.9.4.10.4.7 UTILIZATION OF OTHER ELEMENTS

UTILIZATION OF OTHER ELEMENTS IN TOP LEVEL COMPONENT:

The following tables describe the elements used by this part but defined elsewhere in the parent top level component:

Subprograms and task entries:

The following table summarizes the subroutines and task entries required by this part and defined in the package body of Unbounded_FIFO_Buffer:

Name	Type	Description
New_Node	function	Returns a node to the calling routine; will get a node from the available space list if possible, otherwise will allocate a new node

Data types:

The following table summarizes the types required by this part and defined as generic parameters to the Abstract_Data_Structures. Unbounded_FIFO_Buffer package:

Name	Type	Description
Elements	private	User defined type of data contained in the buffer

The following table summarizes the types required by this part and defined in the private portion of the part's package specification:

Name	Type	Range	Description
Nodes	record	N/A	A single entity in the buffer; contains data and a pointer to the next node
Pointers	access	N/A	Points to a node in the buffer
Buffers	record	N/A	Record containing the value of the current length, head, and tail of the buffer

Exceptions:

The following table summarizes the exceptions required by this part and defined in the package specification of Abstract_Data_Structures. Unbounded_FIFO_-Buffer:

Name	Description
Buffer_Not_Initialized	Raised if an attempt is made to use an uninitialized buffer

The following table describes the exceptions required by this part and defined in the Ada predefined package STANDARD:

Name	Description
STORAGE_ERROR	Raised when an attempt is made to dynamically allocate more memory than is available

3.3.7.1.9.4.10.4.8 LIMITATIONS

The following table describes the exceptions raised by this part:

Name	When/Why Raised
Storage_Error	Raised if an attempt is made to allocate memory when no more is available
Buffer_Not_Initialized	Raised if an attempt is made to use an uninitialized buffer

3.3.7.1.9.4.10.5 RETRIEVE_ELEMENT UNIT DESIGN

This procedure retrieves the oldest element from the FIFO buffer, places the spare node on the available space list, and updates the status of the FIFO buffer.

If the buffer has not been initialized, a Buffer_Not_Initialized exception is raised.

If the buffer is empty, a Buffer_Empty exception is raised.

3.3.7.1.9.4.10.5.1 REQUIREMENTS ALLOCATION

This part meets CAMP requirement R164.

3.3.7.1.9.4.10.5.2 LOCAL ENTITIES DESIGN

None.

3.3.7.1.9.4.10.5.3 INPUT/OUTPUT

FORMAL PARAMETERS:

The following table describes this part's formal parameters:

Name	Type	Mode	Description
Buffer	Buffers	in out	FIFO buffer being accessed
Old_Element	Elements	out	Element retrieved from the buffer

3.3.7.1.9.4.10.5.4 LOCAL DATA

Data objects:

The following table describes the objects maintained local to this part:

Name	Type	Description
This_Node	Pointers	Node to be placed in the available space list

3.3.7.1.9.4.10.5.5 PROCESS CONTROL

Not applicable.

3.3.7.1.9.4.10.5.6 PROCESSING

The following describes the processing performed by this part:

```
   procedure Retrieve_Element (Buffer      : in out Buffers;
                               Old_Element :    out Elements) is

--    ---------------------
--    --declaration section
--    ---------------------

      Current_Length : INTEGER  renames Buffer.Current_Length;
      Head           : Pointers renames Buffer.Head;

      This_Node      : Pointers;

-- ---------------------------------
-- --begin procedure Retrieve_Element
-- ---------------------------------

   begin

--    --make sure an element is available
      if Current_Length = -1 then
         raise Buffer_Not_Initialized;
```

```
      elsif Current_Length = 0 then
         raise Buffer_Empty;
      end if;

--    --save dummy node in the available space list
      This_Node := Head;
      Head      := Head.Next;
      Save_Node (Saved_Node => This_Node);

--    --retrieve element (its node becomes the new dummy node)
      Old_Element := Head.Data;

--    --update buffer status
      Current_Length := Current_Length - 1;

   end Retrieve_Element ;
```

3.3.7.1.9.4.10.5.7 UTILIZATION OF OTHER ELEMENTS

UTILIZATION OF OTHER ELEMENTS IN TOP LEVEL COMPONENT:

The following tables describe the elements used by this part but defined elsewhere in the parent top level component:

Subprograms and task entries:

The following table summarizes the subroutines and task entries required by this part and defined in the package body of Unbounded_FIFO_Buffer:

Name	Type	Description
Save_Node	procedure	Handles placing a node in the available space list

Data types:

The following table summarizes the types required by this part and defined as generic parameters to the Abstract_Data_Structures. Unbounded_FIFO_Buffer package:

Name	Type	Description
Elements	private	User defined type of data contained in the buffer

The following table summarizes the types required by this part and defined in the private portion of the part's package specification:

Name	Type	Range	Description
Nodes	record	N/A	A single entity in the buffer; contains data and a pointer to the next node
Pointers	access	N/A	Points to a node in the buffer
Buffers	record	N/A	Record containing the value of the current length, head, and tail of the buffer

Exceptions:

The following table summarizes the exceptions required by this part and defined in the package specification of Abstract_Data_Structures. Unbounded_FIFO_-Buffer:

Name	Description
Buffer_Empty	Error condition raised if an attempt is made to look at or retrieve elements from an empty buffer
Buffer_Not_Initialized	Raised if an attempt is made to use an uninitialized buffer

3.3.7.1.9.4.10.5.8 LIMITATIONS

The following table describes the exceptions raised by this part:

Name	When/Why Raised
Buffer_Empty	Raised if an attempt is made to access an empty buffer
Buffer_Not_Initialized	Raised if an attempt is made to use an uninitialized buffer

3.3.7.1.9.4.10.6 PEEK UNIT DESIGN

This function returns the oldest element in the FIFO buffer.

If the buffer has not been initialized, a Buffer_Not_Initialized exception is raised.

If the buffer is empty, a Buffer_Empty exception is raised.

3.3.7.1.9.4.10.6.1 REQUIREMENTS ALLOCATION

This part meets CAMP requirement R164.

3.3.7.1.9.4.10.6.2 LOCAL ENTITIES DESIGN

None.

3.3.7.1.9.4.10.6.3 INPUT/OUTPUT

FORMAL PARAMETERS:

The following table describes this part's formal parameters:

Name	Type	Mode	Description
Buffer	Buffers	in out	FIFO buffer being accessed

3.3.7.1.9.4.10.6.4 LOCAL DATA

None.

3.3.7.1.9.4.10.6.5 PROCESS CONTROL

Not applicable.

3.3.7.1.9.4.10.6.6 PROCESSING

The following describes the processing performed by this part:

```
   function Peek (Buffer : in  Buffers) return Elements is

--    ---------------------
--    --declaration section
--    ---------------------

      Current_Length : INTEGER  renames Buffer.Current_Length;
      Head           : Pointers renames Buffer.Head;

-- ---------------------
-- --begin function Peek
-- ---------------------

   begin

--    --make sure something is there to look at
      if Current_Length = -1 then
         raise Buffer_Not_Initialized;
      elsif Current_Length = 0 then
         raise Buffer_Empty;
      end if;

      return Head.Next.Data;

   end Peek ;
```

3.3.7.1.9.4.10.6.7 UTILIZATION OF OTHER ELEMENTS

UTILIZATION OF OTHER ELEMENTS IN TOP LEVEL COMPONENT:

The following tables describe the elements used by this part but defined elsewhere in the parent top level component:

Data types:

The following table summarizes the types required by this part and defined as generic parameters to the Abstract_Data_Structures. Unbounded_FIFO_Buffer package:

Name	Type	Description
Elements	private	User defined type of data contained in the buffer

The following table summarizes the types required by this part and defined in the private portion of the part's package specification:

Name	Type	Range	Description
Nodes	record	N/A	A single entity in the buffer; contains data and a pointer to the next node
Pointers	access	N/A	Points to a node in the buffer
Buffers	record	N/A	Record containing the value of the current length, head, and tail of the buffer

Exceptions:

The following table summarizes the exceptions required by this part and defined in the package specification of Abstract_Data_Structures. Unbounded_FIFO_-Buffer:

Name	Description
Buffer_Empty	Error condition raised if an attempt is made to look at or retrieve elements from an empty buffer
Buffer_Not_Initialized	Raised if an attempt is made to use an uninitialized buffer

3.3.7.1.9.4.10.6.8 LIMITATIONS

The following table describes the exceptions raised by this part:

Name	When/Why Raised
Buffer_Empty	Raised if an attempt is made to access an empty buffer
Buffer_Not_Initialized	Raised if an attempt is made to use an uninitialized buffer

3.3.7.1.9.4.10.7 BUFFER_STATUS UNIT DESIGN

This function returns the status of the buffer based on the following algorithm:

if buffer has never been initialized then status is uninitialized elsif buffer has no nodes in it then status is empty else status is available

3.3.7.1.9.4.10.7.1 REQUIREMENTS ALLOCATION

This part meets CAMP requirement R164.

3.3.7.1.9.4.10.7.2 LOCAL ENTITIES DESIGN

None.

3.3.7.1.9.4.10.7.3 INPUT/OUTPUT

FORMAL PARAMETERS:

The following table describes this part's formal parameters:

Name	Type	Mode	Description
Buffer	Buffers	in out	FIFO buffer being accessed

3.3.7.1.9.4.10.7.4 LOCAL DATA

None.

3.3.7.1.9.4.10.7.5 PROCESS CONTROL

Not applicable.

3.3.7.1.9.4.10.7.6 PROCESSING

The following describes the processing performed by this part:

```
function Buffer_Status (Buffer : in Buffers) return Buffer_Statuses is
```

```
--    ---------------------
--    --declaration section
--    ---------------------

      Current_Length : INTEGER  renames Buffer.Current_Length;
      Status         : Buffer_Statuses;

-- ------------------------------
-- --begin function Buffer_Status
-- ------------------------------

   begin

      if Current_Length = -1 then
         Status := Uninitialized;

      elsif Current_Length = 0 then
         Status := Empty;

      else
         Status := Available;

      end if;

      return Status;

   end Buffer_Status ;
```

3.3.7.1.9.4.10.7.7 UTILIZATION OF OTHER ELEMENTS

UTILIZATION OF OTHER ELEMENTS IN TOP LEVEL COMPONENT:

The following tables describe the elements used by this part but defined elsewhere in the parent top level component:

Data types:

The following table summarizes the types required by this part and defined in the package specification of Abstract_Data_Structures. Unbounded_FIFO_Buffer:

Name	Type	Range	Description
Buffer_ Statuses	discrete type	Empty, Available, Uninitialized	Used to indicate the status of the buffer

The following table summarizes the types required by this part and defined in the private portion of the part's package specification:

Name	Type	Range	Description
Buffers	record	N/A	Record containing the value of the current length, head, and tail of the buffer

3.3.7.1.9.4.10.7.8 LIMITATIONS

None.

3.3.7.1.9.4.10.8 BUFFER_LENGTH UNIT DESIGN

This function returns the length of the current buffer.

If the buffer has not been initialized, a Buffer_Not_Initialized exception is raised.

3.3.7.1.9.4.10.8.1 REQUIREMENTS ALLOCATION

This part meets CAMP requirement R164.

3.3.7.1.9.4.10.8.2 LOCAL ENTITIES DESIGN

None.

3.3.7.1.9.4.10.8.3 INPUT/OUTPUT

FORMAL PARAMETERS:

The following table describes this part's formal parameters:

Name	Type	Mode	Description
Buffer	Buffers	in out	FIFO buffer being accessed

3.3.7.1.9.4.10.8.4 LOCAL DATA

None.

3.3.7.1.9.4.10.8.5 PROCESS CONTROL

Not applicable.

3.3.7.1.9.4.10.8.6 PROCESSING

The following describes the processing performed by this part:

```
   function Buffer_Length (Buffer : in Buffers) return NATURAL is

--    ---------------------
--    --declaration section
--    ---------------------

      Current_Length : INTEGER  renames Buffer.Current_Length;

-- -----------------------------
-- --begin function Buffer_Length
-- -----------------------------

   begin

--    --make sure the buffer has a length
      if Current_Length = -1 then
         raise Buffer_Not_Initialized;
      end if;

      return Current_Length;

   end Buffer_Length ;
```

3.3.7.1.9.4.10.8.7 UTILIZATION OF OTHER ELEMENTS

UTILIZATION OF OTHER ELEMENTS IN TOP LEVEL COMPONENT:

The following tables describe the elements used by this part but defined elsewhere in the parent top level component:

Data types:

The following table summarizes the types required by this part and defined in the private portion of the part's package specification:

Name	Type	Range	Description
Buffers	record	N/A	Record containing the value of the current length, head, and tail of the buffer

Exceptions:

The following table summarizes the exceptions required by this part and defined in the package specification of Abstract_Data_Structures. Unbounded_FIFO_-Buffer:

Name	Description
Buffer_Not_Initialized	Raised if an attempt is made to use an uninitialized buffer

3.3.7.1.9.4.10.8.8 LIMITATIONS

The following table describes the exceptions raised by this part:

Name	When/Why Raised
Buffer_Not_Initialized	Raised if an attempt is made to use an uninitialized buffer

3.3.7.1.9.4.10.9 DOT_NEXT UNIT DESIGN

Given an input pointer P, this function returns the value of P.Next.

3.3.7.1.9.4.10.9.1 REQUIREMENTS ALLOCATION

None.

3.3.7.1.9.4.10.9.2 LOCAL ENTITIES DESIGN

None.

3.3.7.1.9.4.10.9.3 INPUT/OUTPUT

FORMAL PARAMETERS:

The following table describes this part's formal parameters:

Name	Type	Mode	Description
Ptr	Pointers	in	Pointer to the node whose "next" entry is to be returned

3.3.7.1.9.4.10.9.4 LOCAL DATA

None.

3.3.7.1.9.4.10.9.5 PROCESS CONTROL

Not applicable.

3.3.7.1.9.4.10.9.6 PROCESSING

The following describes the processing performed by this part:

```
function Dot_Next (Ptr : in Pointers) return Pointers is
begin
   return Ptr.Next;
end Dot_Next;
```

3.3.7.1.9.4.10.9.7 UTILIZATION OF OTHER ELEMENTS

UTILIZATION OF OTHER ELEMENTS IN TOP-LEVEL COMPONENT:

The following tables describe the elements used by this part but defined elsewhere in the parent top-level component:

Data types:

The following table summarizes the types required by this part and defined in the private portion of the part's package specification:

Name	Type	Range	Description
Nodes	record	N/A	A single entity in the buffer; contains data and a pointer to the next node
Pointers	access	N/A	Points to a node in the buffer
Buffers	record	N/A	Record containing the value of the current length, head, and tail of the buffer

3.3.7.1.9.4.10.9.8 LIMITATIONS

None.

3.3.7.1.9.4.10.10 SET_NEXT UNIT DESIGN

Given an two input pointers, P and Q, this procedure sets P.Next equal to Q.

3.3.7.1.9.4.10.10.1 REQUIREMENTS ALLOCATION

None.

3.3.7.1.9.4.10.10.2 LOCAL ENTITIES DESIGN

None.

3.3.7.1.9.4.10.10.3 INPUT/OUTPUT

FORMAL PARAMETERS:

The following table describes this part's formal parameters:

Name	Type	Mode	Description
Ptr	Pointers	in	Pointer to the node whose "next" entry is to be modified
Ptr_dot_Next	Pointers	in	Value to which Ptr.Next is to be set

3.3.7.1.9.4.10.10.4 LOCAL DATA

None.

3.3.7.1.9.4.10.10.5 PROCESS CONTROL

Not applicable.

3.3.7.1.9.4.10.10.6 PROCESSING

The following describes the processing performed by this part:

```
procedure Set_Next (Ptr          : in Pointers;
                    Ptr_dot_Next : in Pointers) is
begin
   Ptr.Next := Ptr_dot_Next;
end Set_Next;
```

3.3.7.1.9.4.10.10.7 UTILIZATION OF OTHER ELEMENTS

UTILIZATION OF OTHER ELEMENTS IN TOP-LEVEL COMPONENT:

The following tables describe the elements used by this part but defined elsewhere in the parent top-level component:

Data types:

The following table summarizes the types required by this part and defined in the private portion of the part's package specification:

Name	Type	Range	Description
Nodes	record	N/A	A single entity in the buffer; contains data and a pointer to the next node
Pointers	access	N/A	Points to a node in the buffer
Buffers	record	N/A	Record containing the value of the current length, head, and tail of the buffer

3.3.7.1.9.4.10.10.8 LIMITATIONS

None.

3.3.7.1.9.5 NONBLOCKING_CIRCULAR_BUFFER PACKAGE DESIGN (CATALOG #P333-0)

This generic package defines the data type and contains the operations required to perform circular buffering operations on incoming data. These operations are performed in a non-blocking fashion such that if the buffer is full, incoming data will overwrite old data. The head of the buffer always points to a dummy node. The first node following the dummy node contains the next piece of data to be retrieved. The tail always points to where the next element should be added. If the tail points to the element immediately in front of the head, the buffer is empty. If the tail points to the same element as the head, the buffer is full. This is illustrated below.

Empty circular buffer: +-+ <------Head +-+ +-+ +-+ <-----Tail +-+ +-+ +-+ +-+

Full circular buffer: Tail----->+-+ <------Head +-+ +-+ +-+ +-+ +-+ +-+ +-+

The decomposition for this part is the same as that shown in the Top-Level Design Document.

3.3.7.1.9.5.1 REQUIREMENTS ALLOCATION

This part meets CAMP requirement R126.

3.3.7.1.9.5.2 LOCAL ENTITIES DESIGN

None.

3.3.7.1.9.5.3 INPUT/OUTPUT

GENERIC PARAMETERS:

The following generic parameters were previously defined when this part was specified in the package specification of the Abstract_Data_Structures package:

Data types:

The following table summarizes the generic formal types required by this part:

Name	Type	Description
Elements	private	User defined type of data contained in the buffer

Data objects:

The following table summarizes the generic formal objects required by this part:

Name	Type	Value	Description
Initial_ Buffer_Size	POSITIVE	N/A	Maximum number of elements which can be in the buffer at any given time

3.3.7.1.9.5.4 LOCAL DATA

None.

3.3.7.1.9.5.5 PROCESS CONTROL

Not applicable.

3.3.7.1.9.5.6 PROCESSING

The follow. ɜ describes the processing performed by this part:

```
separate (Abstract_Data_Structures)
package body Nonblocking_Circular_Buffer is

end Nonblocking_Circular_Buffer;
```

3.3.7.1.9.5.7 UTILIZATION OF OTHER ELEMENTS

UTILIZATION OF OTHER ELEMENTS IN TOP LEVEL COMPONENT:

The following tables describe the elements used by this part but defined elsewhere in the parent top level component:

Data types:

The following data types were previously defined in this part's package specification:

Name	Type	Range	Description
Buffer_Range	NATURAL subtype	0 .. Buffer_Size	Used to dimension the list of elements
Buffer_Statuses	discrete type	Empty, Available, Full	Used to indicate the status of the buffer

The following table describes the data types defined in the private part of the Abstract_Data_Structures.Nonblocking_Circular_Buffer package:

Name	Type	Range	Description
Lists Buffers	array record	N/A N/A	Array of elements List of data along with relevant information

Data objects:

The following data objects were previously defined in this part's package specification:

Name	Type	Value	Description
Buffer_Size	POSITIVE	Initial_Buffer_Size	Number of usable elements in a buffer

Exceptions:

The following exceptions were previously defined in this part's package specification:

Name	Description
Buffer_Empty	Error condition raised if an attempt is made to look at or retrieve elements from an empty buffer

3.3.7.1.9.5.8 LIMITATIONS

None.

3.3.7.1.9.5.9 LLCSC DESIGN

None.

3.3.7.1.9.5.10 UNIT DESIGN

3.3.7.1.9.5.10.1 CLEAR_BUFFER UNIT DESIGN

This procedure clears a buffer by setting the Head to 0, the Tail to 1, and the length to 0.

3.3.7.1.9.5.10.1.1 REQUIREMENTS ALLOCATION

This part meets CAMP requirement R126.

3.3.7.1.9.5.10.1.2 LOCAL ENTITIES DESIGN

None.

3.3.7.1.9.5.10.1.3 INPUT/OUTPUT

FORMAL PARAMETERS:

The following table describes this part's formal parameters:

Name	Type	Mode	Description
Buffer	Buffers	out	Nonblocking circular buffer being accessed

3.3.7.1.9.5.10.1.4 LOCAL DATA

None.

3.3.7.1.9.5.10.1.5 PROCESS CONTROL

Not applicable.

3.3.7.1.9.5.10.1.6 PROCESSING

The following describes the processing performed by this part:

```
   procedure Clear_Buffer (Buffer : out Buffers) is

--    ---------------------
--    --declaration section
--    ---------------------

      Head           : Buffer_Range renames Buffer.Head;
      Tail           : Buffer_Range renames Buffer.Tail;
      Current_Length : Buffer_Range renames Buffer.Current_Length;

-- ----------------------------
```

```
-- --begin procedure Clear_Buffer
-- ------------------------------

   begin

      Head           := 0;
      Tail           := 1;
      Current_Length := 0;

   end Clear_Buffer ;
```

3.3.7.1.9.5.10.1.7 UTILIZATION OF OTHER ELEMENTS

UTILIZATION OF OTHER ELEMENTS IN TOP LEVEL COMPONENT:

The following tables describe the elements used by this part but defined elsewhere in the parent top level component:

Data types:

The following table summarizes the types required by this part and defined in the package specification of the Nonblocking_Circular_Buffer package:

Name	Type	Range	Description
Buffer_Range	NATURAL subtype	0 .. Buffer_Size	Used to dimension the list of elements

The following table describes the data types defined in the private part of the Abstract_Data_Structures.Nonblocking_Circular_Buffer package:

Name	Type	Range	Description
Buffers	record	N/A	List of data along with relevant information

3.3.7.1.9.5.10.1.8 LIMITATIONS

None.

3.3.7.1.9.5.10.2 ADD_ELEMENT UNIT DESIGN

This procedure adds an element to the end of the buffer, overwriting old data if the buffer is full. If data was overwritten, both the head and tail of the buffer are adjusted to reflect the current status of the buffer. If data was not overwritten, only the tail of the buffer is adjusted.

3.3.7.1.9.5.10.2.1 REQUIREMENTS ALLOCATION

This part meets CAMP requirement R126.

3.3.7.1.9.5.10.2.2 LOCAL ENTITIES DESIGN

None.

3.3.7.1.9.5.10.2.3 INPUT/OUTPUT

FORMAL PARAMETERS:

The following table describes this part's formal parameters:

Name	Type	Mode	Description
Buffer	Buffers	out	Circular buffer being accessed
New_Element	Elements	in	Element to be added to the buffer

3.3.7.1.9.5.10.2.4 LOCAL DATA

None.

3.3.7.1.9.5.10.2.5 PROCESS CONTROL

Not applicable.

3.3.7.1.9.5.10.2.6 PROCESSING

The following describes the processing performed by this part:

```
   procedure Add_Element (New_Element : in     Elements;
                          Buffer      : in out Buffers) is

--    ---------------------
--    --declaration section
--    ---------------------

      Head           : Buffer_Range renames Buffer.Head;
      Tail           : Buffer_Range renames Buffer.Tail;
      Current_Length : Buffer_Range renames Buffer.Current_Length;
      List           : Lists renames Buffer.List;

-- ----------------------------
-- --begin procedure Add_Element
-- ----------------------------

   begin

      List(Tail) := New_Element;
```

```
      if Head = Tail then

--       --buffer was already full and an element was overwritten; therefore,
--       --both head and tail need to be advanced, but Current_Length does
--       --not need to be changed

         if Tail = Buffer_Size then
            Head := 0;
            Tail := 0;
         else
            Head := Head + 1;
            Tail := Tail + 1;
         end if;

      else

--       --buffer was not already full; therefore, the Current_Length needs
--       --to be increment and only the tail needs to be advanced

         if Tail = Buffer_Size then
            Tail := 0;
         else
            Tail := Tail + 1;
         end if;

         Current_Length := Current_Length + 1;

      end if;

   end Add_Element ;
```

3.3.7.1.9.5.10.2.7 UTILIZATION OF OTHER ELEMENTS

UTILIZATION OF OTHER ELEMENTS IN TOP LEVEL COMPONENT:

The following tables describe the elements used by this part but defined elsewhere in the parent top level component:

Data types:

The following table summarizes the types required by this part and defined as generic formal types to the Nonblocking_Circular_Buffer package:

Name	Type	Description
Elements	private	User defined type of data contained in the buffer

The following table summarizes the types required by this part and defined in the package specification of the Nonblocking_Circular_Buffer package:

Name	Type	Range	Description
Buffer_Range	NATURAL subtype	0 .. Buffer_Size	Used to dimension the list of elements

The following table describes the data types defined in the private part of the Abstract_Data_Structures.Nonblocking_Circular_Buffer package:

Name	Type	Range	Description
Buffers	record	N/A	List of data along with relevant information
Lists	array	N/A	Array of elements

Data objects:

The following table summarizes the types required by this part and defined in the package specification of Nonblocking_Circular_Buffer:

Name	Type	Value	Description
Buffer_Size	POSITIVE	Initial_Buffer_Size	Number of usable elements in a buffer

3.3.7.1.9.5.10.2.8 LIMITATIONS

None.

3.3.7.1.9.5.10.3 RETRIEVE_ELEMENT UNIT DESIGN

This procedure returns the first element in the circular buffer.

If there are no elements in the buffer, a Buffer_Empty exception is raised.

3.3.7.1.9.5.10.3.1 REQUIREMENTS ALLOCATION

This part meets CAMP requirement R126.

3.3.7.1.9.5.10.3.2 LOCAL ENTITIES DESIGN

None.

3.3.7.1.9.5.10.3.3 INPUT/OUTPUT

FORMAL PARAMETERS:

The following table describes this part's formal parameters:

Name	Type	Mode	Description
Buffer	Buffers	out	Circular buffer being accessed
Old_Element	Elements	out	Element retrieved from the buffer

3.3.7.1.9.5.10.3.4 LOCAL DATA

None.

3.3.7.1.9.5.10.3.5 PROCESS CONTROL

Not applicable.

3.3.7.1.9.5.10.3.6 PROCESSING

The following describes the processing performed by this part:

```
   procedure Retrieve_Element (Buffer      : in out Buffers;
                               Old_Element :    out Elements) is

--    ---------------------
--    --declaration section
--    ---------------------

      Head           : Buffer_Range renames Buffer.Head;
      Tail           : Buffer_Range renames Buffer.Tail;
      Current_Length : Buffer_Range renames Buffer.Current_Length;
      List           : Lists        renames Buffer.List;

-- ---------------------------------
-- --begin procedure Retrieve_Element
-- ---------------------------------

   begin

--    --make sure there is something there to retrieve
      if Current_Length = 0 then
         raise Buffer_Empty;
      end if;

--    --advance the head to get to the next element to go out
      if Head = Buffer_Size then
         Head := 0;
      else
         Head := Head + 1;
      end if;
```

```
--    --now retrieve the element and update the state of the buffer
      Old_Element := List(Head);
      Current_Length := Current_Length - 1;

   end Retrieve_Element ;
```

3.3.7.1.9.5.10.3.7 UTILIZATION OF OTHER ELEMENTS

UTILIZATION OF OTHER ELEMENTS IN TOP LEVEL COMPONENT:

The following tables describe the elements used by this part but defined elsewhere in the parent top level component:

Data types:

The following table summarizes the types required by this part and defined as generic formal types to the Nonblocking_Circular_Buffer package:

Name	Type	Description
Elements	private	User defined type of data contained in the buffer

The following table summarizes the types required by this part and defined in the package specification of the Nonblocking_Circular_Buffer package:

Name	Type	Range	Description
Buffer_Range	NATURAL subtype	0 .. Buffer_Size	Used to dimension the list of elements

The following table describes the data types defined in the private part of the Abstract_Data_Structures.Nonblocking_Circular_Buffer package:

Name	Type	Range	Description
Buffers	record	N/A	List of data along with relevant information
Lists	array	N/A	Array of elements

Data objects:

The following table summarizes the types required by this part and defined in the package specification of Nonblocking_Circular_Buffer:

Name	Type	Value	Description
Buffer_Size	POSITIVE	Initial_Buffer_Size	Number of usable elements in a buffer

Exceptions:

The following table summarizes the exceptions required by this part and defined in the package specification of Nonblocking_Circular_Buffer:

Name	Description
Buffer_Empty	Error condition raised if an attempt is made to look at or retrieve elements from an empty buffer

3.3.7.1.9.5.10.3.8 LIMITATIONS

The following table describes the exceptions raised by this part:

Name	Description
Buffer_Empty	Error condition raised if an attempt is made to look at or retrieve elements from an empty buffer

3.3.7.1.9.5.10.4 PEEK UNIT DESIGN

This function returns the data contained in the first element in the buffer without changing the state of the buffer (i.e., the element is not removed from the buffer).

If there are no elements in the buffer, a Buffer_Empty exception is raised.

3.3.7.1.9.5.10.4.1 REQUIREMENTS ALLOCATION

This part meets CAMP requirement R126.

3.3.7.1.9.5.10.4.2 LOCAL ENTITIES DESIGN

None.

3.3.7.1.9.5.10.4.3 INPUT/OUTPUT

FORMAL PARAMETERS:

The following table describes this part's formal parameters:

Name	Type	Mode	Description
Buffer	Buffers	out	Circular buffer being accessed

3.3.7.1.9.5.10.4.4 LOCAL DATA

Data objects:

The following table describes the data objects maintained by this part:

Name	Type	Value	Description
Spot	Buffer_Range	N/A	Marks the spot in the buffer containing the element to be looked at

3.3.7.1.9.5.10.4.5 PROCESS CONTROL

Not applicable.

3.3.7.1.9.5.10.4.6 PROCESSING

The following describes the processing performed by this part:

```
   function Peek (Buffer : in Buffers) return Elements is

--    ---------------------
--    --declaration section
--    ---------------------

      Head           : Buffer_Range renames Buffer.Head;
      Current_Length : Buffer_Range renames Buffer.Current_Length;
      List           : Lists        renames Buffer.List;

      Spot           : Buffer_Range;

-- ---------------------
-- --begin function Peek
-- ---------------------

   begin

--    --make sure there is something to peek at
      if Current_Length = 0 then
         raise Buffer_Empty;
      end if;

--    --determine location of desired element
      if Head = Buffer_Size then
```

```
         Spot := 0;
      else
         Spot := Head + 1;
      end if;

--    --return requested element
      return List(Spot);

   end Peek ;
```

3.3.7.1.9.5.10.4.7 UTILIZATION OF OTHER ELEMENTS

UTILIZATION OF OTHER ELEMENTS IN TOP LEVEL COMPONENT:

The following tables describe the elements used by this part but defined elsewhere in the parent top level component:

Data types:

The following table summarizes the types required by this part and defined as generic formal types to the Nonblocking_Circular_Buffer package:

Name	Type	Description
Elements	private	User defined type of data contained in the buffer

The following table summarizes the types required by this part and defined in the package specification of the Nonblocking_Circular_Buffer package:

Name	Type	Range	Description
Buffer_Range	NATURAL subtype	0 .. Buffer_Size	Used to dimension the list of elements

The following table describes the data types defined in the private part of the Abstract_Data_Structures.Nonblocking_Circular_Buffer package:

Name	Type	Range	Description
Buffers	record	N/A	List of data along with relevant information
Lists	array	N/A	Array of elements

Data objects:

The following table summarizes the types required by this part and defined in the package specification of Nonblocking_Circular_Buffer:

Name	Type	Value	Description
Buffer_Size	POSITIVE	Initial_Buffer_Size	Number of usable elements in a buffer

Exceptions:

The following table summarizes the exceptions required by this part and defined in the package specification of Nonblocking_Circular_Buffer:

Name	Description
Buffer_Empty	Error condition raised if an attempt is made to look at or retrieve elements from an empty buffer

3.3.7.1.9.5.10.4.8 LIMITATIONS

The following table describes the exceptions raised by this part:

Name	Description
Buffer_Empty	Error condition raised if an attempt is made to look at or retrieve elements from an empty buffer

3.3.7.1.9.5.10.5 BUFFER_STATUS UNIT DESIGN

This function returns the current status of the buffer according to the following algorithm:

if there are no elements in the buffer then buffer status is empty elsif if the buffer contains the maximum number of elements buffer status is full else buffer status is available end if;

3.3.7.1.9.5.10.5.1 REQUIREMENTS ALLOCATION

This part meets CAMP requirement R126.

3.3.7.1.9.5.10.5.2 LOCAL ENTITIES DESIGN

None.

3.3.7.1.9.5.10.5.3 INPUT/OUTPUT

FORMAL PARAMETERS:

The following table describes this part's formal parameters:

Name	Type	Mode	Description
Buffer	Buffers	out	Circular buffer being accessed

3.3.7.1.9.5.10.5.4 LOCAL DATA

Data objects:

The following table describes the data objects maintained by this part:

Name	Type	Value	Description
Status	Buffer_Statuses	N/A	Current status of the buffer

3.3.7.1.9.5.10.5.5 PROCESS CONTROL

Not applicable.

3.3.7.1.9.5.10.5.6 PROCESSING

The following describes the processing performed by this part:

```
   function Buffer_Status (Buffer : in Buffers) return Buffer_Statuses is

--    ---------------------
--    --declaration section
--    ---------------------

      Current_Length : Buffer_Range renames Buffer.Current_Length;

      Status         : Buffer_Statuses;

-- -----------------------------
-- --begin function Buffer_Status
-- -----------------------------

   begin

      if Current_Length = 0 then
         Status := Empty;
      elsif Current_Length = Buffer_Size then
         Status := Full;
      else
         Status := Available;
```

```
      end if;

      return Status;

   end Buffer_Status ;
```

3.3.7.1.9.5.10.5.7 UTILIZATION OF OTHER ELEMENTS

UTILIZATION OF OTHER ELEMENTS IN TOP LEVEL COMPONENT:

The following tables describe the elements used by this part but defined elsewhere in the parent top level component:

Data types:

The following table summarizes the types required by this part and defined in the package specification of the Nonblocking_Circular_Buffer package:

Name	Type	Range	Description
Buffer_Range	NATURAL subtype	0 .. Buffer_Size	Used to dimension the list of elements
Buffer_Statuses	discrete type	Empty, Available, Full	Used to indicate the status of the buffer

The following table describes the data types defined in the private part of the Abstract_Data_Structures.Nonblocking_Circular_Buffer package:

Name	Type	Range	Description
Buffers	record	N/A	List of data along with relevant information

Data objects:

The following table summarizes the types required by this part and defined in the package specification of Nonblocking_Circular_Buffer:

Name	Type	Value	Description
Buffer_Size	POSITIVE	Initial_Buffer_Size	Number of usable elements in a buffer

3.3.7.1.9.5.10.5.8 LIMITATIONS

None.

3.3.7.1.9.5.10.6 BUFFER_LENGTH UNIT DESIGN

This function returns the current length of the buffer.

3.3.7.1.9.5.10.6.1 REQUIREMENTS ALLOCATION

This part meets CAMP requirement R126.

3.3.7.1.9.5.10.6.2 LOCAL ENTITIES DESIGN

None.

3.3.7.1.9.5.10.6.3 INPUT/OUTPUT

FORMAL PARAMETERS:

The following table describes this part's formal parameters:

Name	Type	Mode	Description
Buffer	Buffers	out	Circular buffer being accessed

3.3.7.1.9.5.10.6.4 LOCAL DATA

None.

3.3.7.1.9.5.10.6.5 PROCESS CONTROL

Not applicable.

3.3.7.1.9.5.10.6.6 PROCESSING

The following describes the processing performed by this part:

```
function Buffer_Length (Buffer : in Buffers) return Buffer_Range is
begin
   return Buffer.Current_Length;
end Buffer_Length ;
```

3.3.7.1.9.5.10.6.7 UTILIZATION OF OTHER ELEMENTS

UTILIZATION OF OTHER ELEMENTS IN TOP LEVEL COMPONENT:

The following tables describe the elements used by this part but defined elsewhere in the parent top level component:

Data types:

The following table summarizes the types required by this part and defined in the package specification of the Nonblocking_Circular_Buffer package:

Name	Type	Range	Description
Buffer_Range	NATURAL subtype	0 .. Buffer_Size	Used to dimension the list of elements

The following table describes the data types defined in the private part of the Abstract_Data_Structures.Nonblocking_Circular_Buffer package:

Name	Type	Range	Description
Buffers	record	N/A	List of data along with relevant information

3.3.7.1.9.5.10.6.8 LIMITATIONS

None.

3.3.7.1.9.6 UNBOUNDED_PRIORITY_QUEUE PACKAGE DESIGN (CATALOG #P334-0)

This generic package defines the data type and contains the operations required to perform priority queueing operations on incoming data. The head of the queue always points to a dummy node. The node following the dummy node contains the element with the highest priority. The tail always points to the element with the lowest priority.

The elements will be ordered in the queue such that: 1) Elements with higher priorities are placed before those with lower priorities. 2) Elements with the same priority are arranged in the queue in a first-in-first-out manner.

A queue must be initialized before it is used. If an attempt is made to use an uninitialized queue, the exception Queue_Not_Initialized will be raised. The Initialized_Queue procedure returns an initialized queue. The Clear_Queue procedure returns the nodes of a queue to the available space list and then returns an initialized queue.

An available space list is maintained local to this part. When this part is elaborated the available space list will have a dummy node plus Initial_Available_Space_Size nodes. When nodes are added to the queue, the Add_Element routine will try to get a node from the available space list before attempting to allocate more memory. When the Retrieve_Element routine is called, the unused node will be returned to the available space list for later use. The memory committed to the available space may be deallocated by calling the Free_Memory procedure.

The decomposition for this part is the same as that shown in the Top-Level Design Document.

3.3.7.1.9.6.1 REQUIREMENTS ALLOCATION

This part meets CAMP requirement R165.

3.3.7.1.9.6.2 LOCAL ENTITIES DESIGN

Data structures:

An available space list is maintain local to this part's package body.

Subprograms:

The following subprograms are contained local to this body:

Name	Type	Description
Free_Node	procedure	Instantiation of UNCHECKED_DEALLOCATION
Dot_Next	function	Given a pointer P, this function returns the value of P.Next
Set_Next	procedure	Given two points P & Q, this procedure sets P.Next = Q

The following subprograms are contained in this part as a result of renaming operations on identically named routines contained in the locally instantiated Available_Space_Operations package.

Name	Type	Description
New_Node	function	Returns a node to the calling routine; will get a node from the available space list if possible, otherwise will allocate a new node
Save_Node	procedure	Handles placing a node in the available space list
Save_List	procedure	Handles placing a list of nodes in the available space list

This package body contains code to initialize the Available_Space_List. This code is executed when the package is elaborated. If the generic formal object Initial_Available_Space_Size is greater than or equal to 1, this routine then places the requested number of nodes (in addition to the dummy node) in the available space list.

3.3.7.1.9.6.3 INPUT/OUTPUT

GENERIC PARAMETERS:

The following generic parameters were previously defined when this part was specified in the package specification of the Abstract_Data_Structures package:

Data types:

The following table summarizes the generic formal types required by this part:

Name	Type	Description
Elements	private	User defined type of data contained in the queue
Priorities	private	User defined type determining the priority of the node

Data objects:

The following table summarizes the generic formal objects required by this part:

Name	Type	Description
Initial_Available_Space_Size	NATURAL	Number of available nodes to be initially placed in the available space list

Subprograms:

The following table summarizes the generic formal subroutines required by this part:

Name	Type	Description
">"	function	Used to determine ordering of priorities

3.3.7.1.9.6.4 LOCAL DATA

Data objects:

The following table summarizes the data objects defined by this part as the result of renames:

Name	Type	Value	Description
Available_Length	INTEGER	Available_Space.Current_Length	Indicates the current length of the available space list
Available_Head	Pointers	Available_Space.Head	Points to the head node in the available space list
Available_Tail	Pointers	Available_Space.Tail	Points to the tail node in the available space list

3.3.7.1.9.6.5 PROCESS CONTROL

Not applicable.

3.3.7.1.9.6.6 PROCESSING

The following describes the processing performed by this part:

```
with UNCHECKED_DEALLOCATION;
separate (Abstract_Data_Structures)
package body Unbounded_Priority_Queue is

-- ---------------------
-- --declaration section
-- ---------------------

-- --this pointers is accessed ONLY when setting up the Available_Space
   Initial_Head    : Pointers := new Nodes;

   Available_Space : Queues := (Current_Length => 0,
                                Head           => Initial_Head,
                                Tail           => Initial_Head);

   Available_Length : INTEGER  renames Available_Space.Current_Length;
   Available_Head   : Pointers renames Available_Space.Head;
   Available_Tail   : Pointers renames Available_Space.Tail;

   procedure Free is new UNCHECKED_DEALLOCATION
                          (Object => Nodes,
                           Name   => Pointers);

   procedure Free_Node (Which_Node : in out Pointers)
             renames Free;

   function Dot_Next (Ptr : in Pointers) return Pointers;

   procedure Set_Next (Ptr          : in Pointers;
                       Ptr_dot_Next : in Pointers);

   package Available_Space_Operations is new
           Available_Space_List_Operations
              (Nodes                => Nodes,
               Pointers             => Pointers,
               Available_Length     => Available_Length,
               Available_Head       => Available_Head,
               Available_Tail       => Available_Tail);

   function New_Node return Pointers
            renames Available_Space_Operations.New_Node;

   procedure Save_Node (Saved_Node : in Pointers)
            renames Available_Space_Operations.Save_Node;

   procedure Save_List (Saved_Head : in Pointers;
                        Saved_Tail : in Pointers;
                        Node_Count : in POSITIVE)
```

```
            renames Available_Space_Operations.Save_List;


-----------------------------------------
--begin package Unbounded_Priority_Queue
--(see header for package body for details)
-----------------------------------------
begin

-- --set up available space list if one is desired
   if Initial_Available_Space_Size > 0 then

      Add_Nodes_To_Available_Space_List:
         for I in 1..Initial_Available_Space_Size loop
            Available_Tail.Next := NEW Nodes;
            Available_Tail      := Available_Tail.Next;
         end loop Add_Nodes_to_Available_Space_List;

      Available_Length    := Initial_Available_Space_Size;

   end if;

end Unbounded_Priority_Queue;
```

3.3.7.1.9.6.7 UTILIZATION OF OTHER ELEMENTS

The following library units are with'd by this part:

1. Unchecked_Deallocation

Subprograms and task entries:

The following table describes the subroutines required by this part:

Name	Type	Source	Description
Unchecked_Deallocation	generic function	N/A	Used to deallocate memory

Exceptions:

The following table describes the exceptions required by this part and defined in the Ada predefined package STANDARD:

Name	Description
Storage_Error	Raised when an attempt is made to dynamically allocate more memory than is available

UTILIZATION OF OTHER ELEMENTS IN TOP LEVEL COMPONENT:

The following tables describe the elements used by this part but defined elsewhere in the parent top level component:

Packages:

The following table describes the packages required by this part and specified in the package body of the Abstract_Data_Structures package:

Name	Type	Description
Available_Space_List_Operations	generic package	Contains the routines required to retrieve a node from and place a node in the available space list

Data types:

The following data types were previously defined in this part's package specification:

Name	Type	Range	Description
Queue_Statuses	discrete type	Empty, Available, Uninitialized	Used to indicate the status of the queue

The following data types were previously defined in the private portion of this part's package specification:

Name	Type	Range	Description
Nodes	record	N/A	A single entity in the queue; contains data and a pointer to the next node
Pointers	access	N/A	Points to a node in the queue
Queues	record	N/A	Record containing the value of the current length, head, and tail of the queue

Exceptions:

The following exceptions were previously defined in this part's package specification:

Name	Description
Queue_Empty	Error condition raised if an attempt is made to look at or retrieve elements from an empty queue
Queue_Not_Initialized	Indicates an attempt was made to use an uninitialized queue

3.3.7.1.9.6.8 LIMITATIONS

The following table describes the exceptions raised by this part:

Name	When/Why Raised
Storage_Error	Raised during elaboration of this package if an attempt is made to allocate memory when no more is available

3.3.7.1.9.6.9 LLCSC DESIGN

None.

3.3.7.1.9.6.10 UNIT DESIGN

3.3.7.1.9.6.10.1 INITIALIZE UNIT DESIGN

This procedure initializes a queue by placing a dummy node in it, pointing the head and the tail to the dummy node, and setting the length to 0.

3.3.7.1.9.6.10.1.1 REQUIREMENTS ALLOCATION

This part meets CAMP requirement R165.

3.3.7.1.9.6.10.1.2 LOCAL ENTITIES DESIGN

None.

3.3.7.1.9.6.10.1.3 INPUT/OUTPUT

FORMAL PARAMETERS:

The following table describes this part's formal parameters:

Name	Type	Mode	Description
Queue	Queues	in out	Unbounded priority queue being manipulated

3.3.7.1.9.6.10.1.4 LOCAL DATA

None.

3.3.7.1.9.6.10.1.5 PROCESS CONTROL

Not applicable.

3.3.7.1.9.6.10.1.6 PROCESSING

The following describes the processing performed by this part:

```
   procedure Initialize (Queue : in out Queues) is

--    ---------------------
--    --declaration section
--    ---------------------

      Current_Length : INTEGER  renames Queue.Current_Length;
      Head           : Pointers renames Queue.Head;
      Tail           : Pointers renames Queue.Tail;

-- ----------------------------
-- --begin procedure Initialize
-- ----------------------------

   begin

      if Current_Length = -1 then

--       --handle an uninitialized queue
         Head := New_Node;
         Tail := Head;
         Current_Length := 0;

      elsif Current_Length > 0 then

--       --handle a queue that has something in it
         Clear_Queue(Queue => Queue);

      else

--       --current length = 0 so it is already initialized
         NULL;

      end if;

   end Initialize ;
```

3.3.7.1.9.6.10.1.7 UTILIZATION OF OTHER ELEMENTS

UTILIZATION OF OTHER ELEMENTS IN TOP LEVEL COMPONENT:

The following tables describe the elements used by this part but defined elsewhere in the parent top level component:

Subprograms and task entries:

The following table summarizes the subroutines and task entries required by this part and defined in the package specification of Unbounded_Priority_Queue:

Name	Type	Description
Clear_ Queue	procedure	Returns all the nodes in a queue to the available space list

The following table summarizes the subroutines and task entries required by this part and defined in the package body of Unbounded_Priority_Queue:

Name	Type	Description
New_Node	function	Returns a node to the calling routine; will get a node from the available space list if possible, otherwise will allocate a new node

Data types:

The following table describes the data types required by this part and defined in the private portion of the Abstract_Data_Structures.Unbounded_Priority_Queue package:

Name	Type	Range	Description
Nodes	record	N/A	A single entity in the queue; contains data and a pointer to the next node
Pointers	access	N/A	Points to a node in the queue
Queues	record	N/A	Record containing the value of the current length, head, and tail of the queue

3.3.7.1.9.6.10.1.8 LIMITATIONS

None.

3.3.7.1.9.6.10.2 CLEAR_QUEUE UNIT DESIGN

This procedure removes the nodes from a queue and places them in an available space list.

The Queue_Not_Initialized exception is raised if this routine is called with an uninitialized queue.

3.3.7.1.9.6.10.2.1 REQUIREMENTS ALLOCATION

This part meets CAMP requirement R165.

3.3.7.1.9.6.10.2.2 LOCAL ENTITIES DESIGN

None.

3.3.7.1.9.6.10.2.3 INPUT/OUTPUT

FORMAL PARAMETERS:

The following table describes this part's formal parameters:

Name	Type	Mode	Description
Queue	Queues	in out	Unbounded priority queue being manipulated

3.3.7.1.9.6.10.2.4 LOCAL DATA

Data objects:

The following table describes the data objects maintained local to this part:

Name	Type	Value	Description
This_Node	Pointers	N/A	Points to the node to be returned to the available space list

3.3.7.1.9.6.10.2.5 PROCESS CONTROL

Not applicable

3.3.7.1.9.6.10.2.6 PROCESSING

The following describes the processing performed by this part:

```
   procedure Clear_Queue (Queue : in out Queues) is

--    ----------  --------
--    --declaration section
--    ---------------------

      Current_Length : INTEGER  renames Queue.Current_Length;
      Head           : Pointers renames Queue.Head;
      Tail           : Pointers renames Queue.Tail;

      This_Node      : Pointers;

-- -----------------------------
-- --begin procedure Clear_Queue
-- -----------------------------
```

```
   begin

--     --make sure this is an initialized queue
       if Current_Length = -1 then

          raise Queue_Not_Initialized;

       elsif Current_Length > 0 then

--        --placed nodes in the available space list
          Save_List (Saved_Head => Head.Next,
                     Saved_Tail => Tail,
                     Node_Count => Current_Length);

--        --reinitialize queue variables
          Current_Length := 0;
          Head.Next      := NULL;
          Tail           := Head;

       end if;

   end Clear_Queue ;
```

3.3.7.1.9.6.10.2.7 UTILIZATION OF OTHER ELEMENTS

UTILIZATION OF OTHER ELEMENTS IN TOP LEVEL COMPONENT:

The following tables describe the elements used by this part but defined elsewhere in the parent top level component:

Subprograms and task entries:

The following table summarizes the subroutines and task entries required by this part and defined in the package body of Unbounded_Priority_Queue:

Save_List	procedure	Handles placing a list of nodes in the available space list

Data types:

The following table describes the data types required by this part and defined in the private portion of the Abstract_Data_Structures.Unbounded_Priority_Queue package:

Name	Type	Range	Description
Nodes	record	N/A	A single entity in the queue; contains data and a pointer to the next node
Pointers	access	N/A	Points to a node in the queue
Queues	record	N/A	Record containing the value of the current length, head, and tail of the queue

Exceptions:

The following table summarizes the exceptions required by this part and defined elsewhere in the package specification of Abstract_Data_Structures.Unbounded_-Priority_Queue:

Name	Description
Queue_Empty	Error condition raised if an attempt is made to look at or retrieve elements from an empty queue

3.3.7.1.9.6.10.2.8 LIMITATIONS

The following table describes the exceptions raised by this part:

Name	When/Why Raised
Queue_Not_Initialized	Raised if an attempt is made to manipulate an uninitialized queue

3.3.7.1.9.6.10.3 FREE_MEMORY UNIT DESIGN

This procedure deallocates the memory taken up by the available space list.

3.3.7.1.9.6.10.3.1 REQUIREMENTS ALLOCATION

This part meets CAMP requirement R165.

3.3.7.1.9.6.10.3.2 LOCAL ENTITIES DESIGN

None.

3.3.7.1.9.6.10.3.3 INPUT/OUTPUT

None.

3.3.7.1.9.6.10.3.4 LOCAL DATA

Data objects:

The following table describes the data objects maintained by this part:

Name	Type	Value	Description
Node_to_be_Freed	Pointers	N/A	Points to the node to be deallocated

3.3.7.1.9.6.10.3.5 PROCESS CONTROL

Not applicable.

3.3.7.1.9.6.10.3.6 PROCESSING

The following describes the processing performed by this part:

```
   procedure Free_Memory is

--     ---------------------
--     --declaration section
--     ---------------------

       Node_to_be_Freed : Pointers;

-- ----------------------------
-- --begin procedure Free_Memory
-- ----------------------------

   begin

      Clear_Out_Available_Space_List:
         while Available_Head /= Available_Tail loop
            Node_To_Be_Freed := Available_Head;
            Available_Head   := Available_Head.Next;
            Free_Node (Which_Node => Node_to_be_Freed);
         end loop Clear_Out_Available_Space_List;

      Available_Length := 0;

   end Free_Memory ;
```

3.3.7.1.9.6.10.3.7 UTILIZATION OF OTHER ELEMENTS

UTILIZATION OF OTHER ELEMENTS IN TOP LEVEL COMPONENT:

The following tables describe the elements used by this part but defined elsewhere in the parent top level component:

Subprograms and task entries:

The following table summarizes the subroutines and task entries required by this part and defined in the package body of Unbounded_Priority_Queue:

Name	Type	Description
Free_Node	procedure	Instantiation of UNCHECKED_DEALLOCATION

Data types:

The following table describes the data types required by this part and defined in the private portion of the Abstract_Data_Structures.Unbounded_Priority_Queue package:

Name	Type	Range	Description
Nodes	record	N/A	A single entity in the queue; contains data and a pointer to the next node
Pointers	access	N/A	Points to a node in the queue
Queues	record	N/A	Record containing the value of the current length, head, and tail of the queue

Data objects:

The following table summarizes the objects required by this part and defined in the package body of Abstract_Data_Structures. Unbounded_Priority_Queue:

Name	Type	Value	Description
Available_Length	INTEGER	Available_Space.Current_Length	Indicates the current length of the available space list
Available_Head	Pointers	Available_Space.Head	Points to the head node in the available space list
Available_Tail	Pointers	Available_Space.Tail	Points to the tail node in the available space list

3.3.7.1.9.6.10.3.8 LIMITATIONS

None.

3.3.7.1.9.6.10.4 ADD_ELEMENT UNIT DESIGN

This procedure adds an element to the queue. The elements are added such that the new element is added before the first element which has a smaller priority and after all other elements which a greater or equal priority.

The Queue_Empty exception is raised if this routine is called with an empty queue.

The Queue_Not_Initialized exception is raised if this routine is called with an uninitialized queue.

The Storage_Error exception is raised if a call to this routine requires memory to be dynamically allocated when no more memory is available.

3.3.7.1.9.6.10.4.1 REQUIREMENTS ALLOCATION

This part meets CAMP requirement R165.

3.3.7.1.9.6.10.4.2 LOCAL ENTITIES DESIGN

None.

3.3.7.1.9.6.10.4.3 INPUT/OUTPUT

FORMAL PARAMETERS:

The following table describes this part's formal parameters:

Name	Type	Mode	Description
New_Element	Elements	in	Element to be placed in the queue
New_Priority	Priorities	in	Priority of the element to be placed in the queue
Queue	Queues	in out	Unbounded priority queue being manipulated

3.3.7.1.9.6.10.4.4 LOCAL DATA

Data objects:

The following table describes the data objects maintained by this part:

Name	Type	Value	Description
Before	Pointers	N/A	Points to the element which will go before the new element
Here	Pointers	N/A	Points to the node to be added to the queue

3.3.7.1.9.6.10.4.5 PROCESS CONTROL

Not applicable.

3.3.7.1.9.6.10.4.6 PROCESSING

The following describes the processing performed by this part:

```
procedure Add_Element (New_Element  : in     Elements;
                       New_Priority : in     Priorities;
```

```
                          Queue        : in out Queues) is

--    ---------------------
--    --declaration section
--    ---------------------

      Current_Length : INTEGER  renames Queue.Current_Length;
      Head           : Pointers renames Queue.Head;
      Tail           : Pointers renames Queue.Tail;

      Before         : Pointers;
      Here           : Pointers;

-- ----------------------------
-- --begin procedure Add_Element
-- ----------------------------

   begin

--    --make sure queue has been initialized
      if Current_Length = -1 then
         raise Queue_Not_Initialized;
      end if;

--    --find the nodes which are to go before and after the new element
      Before := Head;
      loop
         exit when (Before = Tail) or else
                   (New_Priority > Before.Next.Priority);
         Before := Before.Next;
      end loop;

--    --now get a new node
      Here := New_Node;

--    --set up the new node
      Here.Priority := New_Priority;
      Here.Data     := New_Element;
      Here.Next     := Before.Next;
      Before.Next   := Here;

--    --readjust the tail, if required
      if Before = Tail then
         Tail := Here;
      end if;

--    --now adjust the queue
      Current_Length := Current_Length + 1;

   end Add_Element ;
```

3.3.7.1.9.6.10.4.7 UTILIZATION OF OTHER ELEMENTS

UTILIZATION OF OTHER ELEMENTS IN TOP LEVEL COMPONENT:

The following tables describe the elements used by this part but defined elsewhere in the parent top level component:

Subprograms and task entries:

The following table summarizes the subroutines and task entries required by this part and defined in the package body of Unbounded_Priority_Queue:

Name	Type	Description
New_Node	function	Returns a node to the calling routine; will get a node from the available space list if possible, otherwise will allocate a new node

The following table summarizes the subroutines and task entries required by this part and defined in the package body of Unbounded_Priority_Queue:

Name	Type	Description
Free_Node	procedure	Instantiation of UNCHECKED_DEALLOCATION

The following table describes the subroutines required by this part and defined as generic formal subroutines to the Abstract_Data_Structures.Unbounded_-Priority_Queue package:

Name	Type	Description
">"	function	Used to determine ordering of priorities

Data types:

The following table describes the data types required by this part and defined in the private portion of the Abstract_Data_Structures.Unbounded_Priority_Queue package:

Name	Type	Range	Description
Nodes	record	N/A	A single entity in the queue; contains data and a pointer to the next node
Pointers	access	N/A	Points to a node in the queue
Queues	record	N/A	Record containing the value of the current length, head, and tail of the queue

Exceptions:

The following table summarizes the exceptions required by this part and defined elsewhere in the package specification of Abstract_Data_Structures.Unbounded_-Priority_Queue:

Name	Description
Storage_ Error	Raised when an attempt is made to dynamically allocate more memory than is available
Queue_Not_ Initialized	Indicates an attempt was made to use an uninitialized queue

3.3.7.1.9.6.10.4.8 LIMITATIONS

The following table describes the exceptions raised by this part:

Name	When/Why Raised
Storage_Error	Raised if an attempt is raised to allocate memory when no more is available
Queue_Not_Initialized	Raised if an attempt is made to manipulate an uninitialized queue

3.3.7.1.9.6.10.5 RETRIEVE_ELEMENT UNIT DESIGN

This procedure returns the first element in the queue.

The Queue_Empty exception is raised if this routine is called with an empty queue.

The Queue_Not_Initialized exception is raised if this routine is called with an uninitialized queue.

3.3.7.1.9.6.10.5.1 REQUIREMENTS ALLOCATION

This part meets CAMP requirement R165.

3.3.7.1.9.6.10.5.2 LOCAL ENTITIES DESIGN

None.

3.3.7.1.9.6.10.5.3 INPUT/OUTPUT

FORMAL PARAMETERS:

The following table describes this part's formal parameters:

Name	Type	Mode	Description
Queue	Queues	in out	Unbounded priority queue being manipulated
Old_Element	Elements	out	Data retrieved from the queue

3.3.7.1.9.6.10.5.4 LOCAL DATA

Data objects:

The following table describes the data objects maintained by this part:

Name	Type	Value	Description
This_Node	Pointers	N/A	Points to the node to be returned to the available space list

3.3.7.1.9.6.10.5.5 PROCESS CONTROL

Not applicable.

3.3.7.1.9.6.10.5.6 PROCESSING

The following describes the processing performed by this part:

```
   procedure Retrieve_Element (Queue       : in out Queues;
                               Old_Element :    out Elements) is

--    --------------------
--    --declaration section
--    --------------------

      Current_Length : INTEGER  renames Queue.Current_Length;
      Head           : Pointers renames Queue.Head;

      This_Node      : Pointers;

-- ---------------------------------
-- --begin procedure Retrieve_Element
-- ---------------------------------

   begin

--    --make sure an element is available
      if Current_Length = -1 then
         raise Queue_Not_Initialized;
      elsif Current_Length = 0 then
         raise Queue_Empty;
      end if;
```

```
--      --save dummy node in the available space list
        This_Node := Head;
        Head      := Head.Next;
        Save_Node (Saved_Node => This_Node);

--      --retrieve element (its node becomes the new dummy node)
        Old_Element := Head.Data;

--      --update queue status
        Current_Length := Current_Length - 1;

    end Retrieve_Element ;
```

3.3.7.1.9.6.10.5.7 UTILIZATION OF OTHER ELEMENTS

UTILIZATION OF OTHER ELEMENTS IN TOP LEVEL COMPONENT:

The following tables describe the elements used by this part but defined elsewhere in the parent top level component:

Subprograms and task entries:

The following table summarizes the subroutines and task entries required by this part and defined in the package body of Unbounded_Priority_Queue:

Name	Type	Description
Save_Node	procedure	Handles placing a node in the available space list

Data types:

The following table describes the data types required by this part and defined in the private portion of the Abstract_Data_Structures.Unbounded_Priority_Queue package:

Name	Type	Range	Description
Nodes	record	N/A	A single entity in the queue; contains data and a pointer to the next node
Pointers	access	N/A	Points to a node in the queue
Queues	record	N/A	Record containing the value of the current length, head, and tail of the queue

Exceptions:

The following table summarizes the exceptions required by this part and defined elsewhere in the package specification of Abstract_Data_Structures.Unbounded_Priority_Queue:

Name	Description
Queue_Empty	Error condition raised if an attempt is made to look at or retrieve elements from an empty queue
Queue_Not_Initialized	Indicates an attempt was made to use an uninitialized queue

3.3.7.1.9.6.10.5.8 LIMITATIONS

The following table describes the exceptions raised by this part:

Name	When/Why Raised
Queue_Empty	Raised if an attempt is made to look at or retrieve from an empty queue
Queue_Not_Initialized	Raised if an attempt is made to manipulate an uninitialized queue

3.3.7.1.9.6.10.6 PEEK UNIT DESIGN

This function returns the value of the first element in the queue, but does not change the state of the queue (i.e., the node is not actually removed from the queue).

The Queue_Empty exception is raised if this routine is called with an empty queue.

The Queue_Not_Initialized exception is raised if this routine is called with an uninitialized queue.

3.3.7.1.9.6.10.6.1 REQUIREMENTS ALLOCATION

This part meets CAMP requirement R165.

3.3.7.1.9.6.10.6.2 LOCAL ENTITIES DESIGN

None.

3.3.7.1.9.6.10.6.3 INPUT/OUTPUT

FORMAL PARAMETERS:

The following table describes this part's formal parameters:

Name	Type	Mode	Description
Queue	Queues	in out	Unbounded priority queue being manipulated

3.3.7.1.9.6.10.6.4 LOCAL DATA

None.

3.3.7.1.9.6.10.6.5 PROCESS CONTROL

Not applicable.

3.3.7.1.9.6.10.6.6 PROCESSING

The following describes the processing performed by this part:

```
   function Peek (Queue : in Queues) return Elements is

--    ---------------------
--    --declaration section
--    ---------------------

      Current_Length : INTEGER  renames Queue.Current_Length;
      Head           : Pointers renames Queue.Head;

-- ---------------------
-- --begin function Peek
-- ---------------------

   begin

--    --make sure something is there to look at
      if Current_Length = -1 then
         raise Queue_Not_Initialized;
      elsif Current_Length = 0 then
         raise Queue_Empty;
      end if;

      return Head.Next.Data;

   end Peek ;
```

3.3.7.1.9.6.10.6.7 UTILIZATION OF OTHER ELEMENTS

UTILIZATION OF OTHER ELEMENTS IN TOP LEVEL COMPONENT:

The following tables describe the elements used by this part but defined elsewhere in the parent top level component:

Data types:

The following table describes the data types required by this part and defined in the private portion of the Abstract_Data_Structures.Unbounded_Priority_Queue package:

Name	Type	Range	Description
Nodes	record	N/A	A single entity in the queue; contains data and a pointer to the next node
Pointers	access	N/A	Points to a node in the queue
Queues	record	N/A	Record containing the value of the current length, head, and tail of the queue

Exceptions:

The following table summarizes the exceptions required by this part and defined elsewhere in the package specification of Abstract_Data_Structures.Unbounded_-Priority_Queue:

Name	Description
Queue_Empty	Error condition raised if an attempt is made to look at or retrieve elements from an empty queue
Queue_Not_Initialized	Indicates an attempt was made to use an uninitialized queue

3.3.7.1.9.6.10.6.8 LIMITATIONS

The following table describes the exceptions raised by this part:

Name	When/Why Raised
Queue_Empty	Raised if an attempt is made to look at or retrieve from an empty queue
Queue_Not_Initialized	Raised if an attempt is made to manipulate an uninitialized queue

3.3.7.1.9.6.10.7 QUEUE_STATUS UNIT DESIGN

This function returns the status of the queue based on the following algorithm:

if the queue has not been initialized then queue status is uninitialized elsif no elements are in the queue then queue status is empty else queue status is available end if;

3.3.7.1.9.6.10.7.1 REQUIREMENTS ALLOCATION

This part meets CAMP requirement R165.

3.3.7.1.9.6.10.7.2 LOCAL ENTITIES DESIGN

None.

3.3.7.1.9.6.10.7.3 INPUT/OUTPUT

FORMAL PARAMETERS:

The following table describes this part's formal parameters:

Name	Type	Mode	Description
Queue	Queues	in out	Unbounded priority queue being manipulated

3.3.7.1.9.6.10.7.4 LOCAL DATA

Data objects:

The following table describes the data objects maintained by this part:

Name	Type	Value	Description
Status	Queue_Statuses	N/A	Status of the queue

3.3.7.1.9.6.10.7.5 PROCESS CONTROL

Not applicable.

3.3.7.1.9.6.10.7.6 PROCESSING

The following describes the processing performed by this part:

```
   function Queue_Status (Queue : in Queues) return Queue_Statuses is

--    ---------------------
--    --declaration section
--    ---------------------

      Current_Length : INTEGER  renames Queue.Current_Length;
      Status         : Queue_Statuses;

-- ----------------------------
-- --begin function Queue_Status
```

```
-- ----------------------------

   begin

      if Current_Length = -1 then
         Status := Uninitialized;
      elsif Current_Length = 0 then
         Status := Empty;
      else
         Status := Available;
      end if;

      return Status;

   end Queue_Status ;
```

3.3.7.1.9.6.10.7.7 UTILIZATION OF OTHER ELEMENTS

UTILIZATION OF OTHER ELEMENTS IN TOP LEVEL COMPONENT:

The following tables describe the elements used by this part but defined elsewhere in the parent top level component:

Data types:

The following table describes the data types required by this part and defined in the package specification of Abstract_Data_Structures.Unbounded_Priority_-Queue:

Name	Type	Range	Description
Queue_ Statuses	discrete type	Empty, Available, Uninitialized	Used to indicate the status of the queue

3.3.7.1.9.6.10.7.8 LIMITATIONS

None.

3.3.7.1.9.6.10.8 QUEUE_LENGTH UNIT DESIGN

This function returns the length of a queue.

The Queue_Not_Initialized exception is raised if this routine is called with an uninitialized queue.

3.3.7.1.9.6.10.8.1 REQUIREMENTS ALLOCATION

This part meets CAMP requirement R165.

3.3.7.1.9.6.10.8.2 LOCAL ENTITIES DESIGN

None.

3.3.7.1.9.6.10.8.3 INPUT/OUTPUT

FORMAL PARAMETERS:

The following table describes this part's formal parameters:

Name	Type	Mode	Description
Queue	Queues	in	Unbounded priority queue being manipulated

3.3.7.1.9.6.10.8.4 LOCAL DATA

None.

3.3.7.1.9.6.10.8.5 PROCESS CONTROL

Not applicable.

3.3.7.1.9.6.10.8.6 PROCESSING

The following describes the processing performed by this part:

```
   function Queue_Length (Queue : in Queues) return NATURAL is

--    ---------------------
--    --declaration section
--    ---------------------

      Current_Length : INTEGER  renames Queue.Current_Length;

-- ----------------------------
-- --begin function Queue_Length
-- ----------------------------

   begin

--    --make sure the queue has a length
      if Current_Length = -1 then
         raise Queue_Not_Initialized;
      end if;

      return Current_Length;

   end Queue_Length ;
```

3.3.7.1.9.6.10.8.7 UTILIZATION OF OTHER ELEMENTS

UTILIZATION OF OTHER ELEMENTS IN TOP LEVEL COMPONENT:

The following tables describe the elements used by this part but defined elsewhere in the parent top level component:

Data types:

The following table describes the data types required by this part and defined in the private portion of the Abstract_Data_Structures.Unbounded_Priority_Queue package:

Name	Type	Range	Description
Nodes	record	N/A	A single entity in the queue; contains data and a pointer to the next node
Pointers	access	N/A	Points to a node in the queue
Queues	record	N/A	Record containing the value of the current length, head, and tail of the queue

Exceptions:

The following table summarizes the exceptions required by this part and defined elsewhere in the package specification of Abstract_Data_Structures.Unbounded_Priority_Queue:

Name	Description
Queue_Not_Initialized	Indicates an attempt was made to use an uninitialized queue

3.3.7.1.9.6.10.8.8 LIMITATIONS

The following table describes the exceptions raised by this part:

Name	When/Why Raised
Queue_Not_Initialized	Raised if an attempt is made to manipulate an uninitialized queue

3.3.7.1.9.6.10.9 DOT_NEXT UNIT DESIGN

Given an input pointer P, this function returns the value of P.Next.

3.3.7.1.9.6.10.9.1 REQUIREMENTS ALLOCATION

None.

3.3.7.1.9.6.10.9.2 LOCAL ENTITIES DESIGN

None.

3.3.7.1.9.6.10.9.3 INPUT/OUTPUT

FORMAL PARAMETERS:

The following table describes this part's formal parameters:

Name	Type	Mode	Description
Ptr	Pointers	in	Pointer to the node whose "next" entry is to be returned

3.3.7.1.9.6.10.9.4 LOCAL DATA

None.

3.3.7.1.9.6.10.9.5 PROCESS CONTROL

Not applicable.

3.3.7.1.9.6.10.9.6 PROCESSING

The following describes the processing performed by this part:

```
function Dot_Next (Ptr : in Pointers) return Pointers is
begin
   return Ptr.Next;
end Dot_Next;
```

3.3.7.1.9.6.10.9.7 UTILIZATION OF OTHER ELEMENTS

UTILIZATION OF OTHER ELEMENTS IN TOP-LEVEL COMPONENT:

The following tables describe the elements used by this part but defined elsewhere in the parent top-level component:

Data types:

The following table summarizes the types required by this part and defined in the private portion of the part's package specification:

Name	Type	Range	Description
Nodes	record	N/A	A single entity in the queue; contains data and a pointer to the next node
Pointers	access	N/A	Points to a node in the queue
Queues	record	N/A	Record containing the value of the current length, head, and tail of the queue

3.3.7.1.9.6.10.9.8 LIMITATIONS

None.

3.3.7.1.9.6.10.10 SET_NEXT UNIT DESIGN

Given an two input pointers, P and Q, this procedure sets P.Next equal to Q.

3.3.7.1.9.6.10.10.1 REQUIREMENTS ALLOCATION

None.

3.3.7.1.9.6.10.10.2 LOCAL ENTITIES DESIGN

None.

3.3.7.1.9.6.10.10.3 INPUT/OUTPUT

FORMAL PARAMETERS:

The following table describes this part's formal parameters:

Name	Type	Mode	Description
Ptr	Pointers	in	Pointer to the node whose "next" entry is to be modified
Ptr_dot_Next	Pointers	in	Value to which Ptr.Next is to be set

3.3.7.1.9.6.10.10.4 LOCAL DATA

None.

3.3.7.1.9.6.10.10.5 PROCESS CONTROL

Not applicable.

3.3.7.1.9.6.10.10.6 PROCESSING

The following describes the processing performed by this part:

```
procedure Set_Next (Ptr          : in Pointers;
                    Ptr_dot_Next : in Pointers) is
begin
   Ptr.Next := Ptr_dot_Next;
end Set_Next;
```

3.3.7.1.9.6.10.10.7 UTILIZATION OF OTHER ELEMENTS

UTILIZATION OF OTHER ELEMENTS IN TOP-LEVEL COMPONENT:

The following tables describe the elements used by this part but defined elsewhere in the parent top-level component:

Data types:

The following table summarizes the types required by this part and defined in the private portion of the part's package specification:

Name	Type	Range	Description
Nodes	record	N/A	A single entity in the queue; contains data and a pointer to the next node
Pointers	access	N/A	Points to a node in the queue
Queues	record	N/A	Record containing the value of the current length, head, and tail of the queue

3.3.7.1.9.6.10.10.8 LIMITATIONS

None.

3.3.7.1.9.7 BOUNDED_STACK PACKAGE DESIGN (CATALOG #P335-0)

This generic package defines the data type and contains the operations required to perform last-in-first-out stacking operations on incoming data. The top of the stack always points to the last element added to the stack and the next element to be removed. When top equals 0, the stack is empty. When top equals Stack_Size, the stack is full.

The decomposition for this part is the same as that shown in the Top-Level Design Document.

3.3.7.1.9.7.1 REQUIREMENTS ALLOCATION

This part meets CAMP require R166.

3.3.7.1.9.7.2 LOCAL ENTITIES DESIGN

None.

3.3.7.1.9.7.3 INPUT/OUTPUT

GENERIC PARAMETERS:

The following generic parameters were previously defined when this part was specified in the package specification of the Abstract_Data_Structures package:

Data types:

The following table summarizes the generic formal types required by this part:

Name	Type	Description
Elements	private	User defined type of data contained in the stack

Data objects:

The following table summarizes the generic formal objects required by this part:

Name	Type	Value	Description
Initial_Stack_Size	POSITIVE	N/A	Maximum number of elements which can be in the stack at any given time

3.3.7.1.9.7.4 LOCAL DATA

None.

3.3.7.1.9.7.5 PROCESS CONTROL

Not applicable.

3.3.7.1.9.7.6 PROCESSING

The following describes the processing performed by this part:

```
separate (Abstract_Data_structures)
package body Bounded_Stack is

end Bounded_Stack;
```

3.3.7.1.9.7.7 UTILIZATION OF OTHER ELEMENTS

UTILIZATION OF OTHER ELEMENTS IN TOP LEVEL COMPONENT:

The following tables describe the elements used by this part but defined elsewhere in the parent top level component:

Data types:

The following data types were previously defined in this part's package specification:

Name	Type	Range	Description
Stack_Length_Range	POSITIVE subtype	1 .. Stack_Size	Used to dimension the list of elements
Stacks	limited private	N/A	List of data along with relevant information
Stack_Statuses	discrete type	Empty, Available, Full	Used to indicate the status of the stack

The following data types were previously defined in the private part of this part's package specification:

Name	Type	Range	Description
Stack_Dimensions_Range	Stack_Dimensions subtype	1 .. 'LAST	Used to dimension the list of elements
Stacks	record	N/A	List of data along with relevant information

Data objects:

The following data objects were previously defined in this part's package specification:

Name	Type	Value	Description
Stack_Size	POSITIVE	Initial_Stack_Size	Number of elements in the stack

Exceptions:

The following exceptions were previously defined in this part's package specification:

Name	Description
Stack_Empty	Error condition raised if an attempt is made to look at or retrieve elements from an empty stack
Stack_Full	Error condition raised if an attempt is made to add elements to a full stack

3.3.7.1.9.7.8 LIMITATIONS

None.

3.3.7.1.9.7.9 LLCSC DESIGN

None.

3.3.7.1.9.7.10 UNIT DESIGN

3.3.7.1.9.7.10.1 CLEAR_STACK UNIT DESIGN

This procedure clears a stack by setting the top to 0.

3.3.7.1.9.7.10.1.1 REQUIREMENTS ALLOCATION

This part meets CAMP requirement R166.

3.3.7.1.9.7.10.1.2 LOCAL ENTITIES DESIGN

None.

3.3.7.1.9.7.10.1.3 INPUT/OUTPUT

FORMAL PARAMETERS:

The following table describes this part's formal parameters:

Name	Type	Mode	Description
Stack	Stacks	in out	Bounded stack being manipulated

3.3.7.1.9.7.10.1.4 LOCAL DATA

None.

3.3.7.1.9.7.10.1.5 PROCESS CONTROL

Not applicable.

3.3.7.1.9.7.10.1.6 PROCESSING

The following describes the processing performed by this part:

```
procedure Clear_Stack (Stack : out Stacks) is

begin

   Stack.Top := 0;

end Clear_Stack ;
```

3.3.7.1.9.7.10.1.7 UTILIZATION OF OTHER ELEMENTS

UTILIZATION OF OTHER ELEMENTS IN TOP LEVEL COMPONENT:

The following tables describe the elements used by this part but defined elsewhere in the parent top level component:

Data types:

The following table summarizes the types required by this part and defined in the package specification of the Abstract_Data_Structures.Bounded_Stack package:

Name	Type	Range	Description
Stack_Length_Range	POSITIVE subtype	1 .. Stack_Size	Used to dimension the list of elements

The following data types were previously defined in the private part of this part's package specification:

Name	Type	Range	Description
Stacks	record	N/A	List of data along with relevant information

3.3.7.1.9.7.10.1.8 LIMITATIONS

None.

3.3.7.1.9.7.10.2 ADD_ELEMENT UNIT DESIGN

This procedure adds an element to the top of the stack.

A Stack_Full exception is raised if this routine is called with a full stack.

3.3.7.1.9.7.10.2.1 REQUIREMENTS ALLOCATION

This part meets CAMP requirement R166.

3.3.7.1.9.7.10.2.2 LOCAL ENTITIES DESIGN

None.

3.3.7.1.9.7.10.2.3 INPUT/OUTPUT

FORMAL PARAMETERS:

The following table describes this part's formal parameters:

Name	Type	Mode	Description
New_Element	Elements	in	Element to be added to the stack
Stack	Stacks	in out	Bounded stack being manipulated

3.3.7.1.9.7.10.2.4 LOCAL DATA

None.

3.3.7.1.9.7.10.2.5 PROCESS CONTROL

Not applicable.

3.3.7.1.9.7.10.2.6 PROCESSING

The following describes the processing performed by this part:

```
   procedure Add_Element (New_Element : in     Elements;
                          Stack       : in out Stacks) is

--    --------------------
--    --declaration section
--    --------------------

      List : Lists              renames Stack.List;
      Top  : Stack_Length_Range renames Stack.Top;

-- ---------------------------
-- --begin procedure Add_Element
```

```
-- ----------------------------

   begin

--    --make sure the stack is not already full
      if Top = Stack_Size then
         raise Stack_Full;
      end if;

--    --add element to the stack
      Top       := Top + 1;
      List(Top) := New_Element;

   end Add_Element ;
```

3.3.7.1.9.7.10.2.7 UTILIZATION OF OTHER ELEMENTS

UTILIZATION OF OTHER ELEMENTS IN TOP LEVEL COMPONENT:

The following tables describe the elements used by this part but defined elsewhere in the parent top level component:

Data types:

The following table summarizes the types required by this part and defined as generic formal parameters to the Abstract_Data_Structures.Bounded_Stack package:

Name	Type	Description
Elements	private	User defined type of data contained in the stack

The following table summarizes the types required by this part and defined in the package specification of the Abstract_Data_Structures.Bounded_Stack package:

Name	Type	Range	Description
Stack_ Length_ Range	POSITIVE subtype	1 .. Stack_Size	Used to dimension the list of elements

The following data types were previously defined in the private part of this part's package specification:

Name	Type	Range	Description
Lists	array	N/A	List of elements
Stacks	record	N/A	List of data along with relevant information

Data objects:

The following table summarizes the objects required by this part and defined in the package specification of Abstract_Data_Structures. Bounded_Stack package:

Name	Type	Value	Description
Stack_Size	POSITIVE	Initial_Stack_Size	Number of elements in the stack

Exceptions:

The following table summarizes the exceptions required by this part and defined in the package specification of Abstract_Data_Structures. Bounded_Stack package:

Name	Description
Stack_Full	Error condition raised if an attempt is made to add elements to a full stack

3.3.7.1.9.7.10.2.8 LIMITATIONS

The following table describes the exceptions raised by this part:

Name	When/Why Raised
Stack_Full	Raised if an attempt is made to add elements to a full stack

3.3.7.1.9.7.10.3 RETRIEVE_ELEMENT UNIT DESIGN

This procedure retrieves the top element from the stack and returns it to the calling routine.

A Stack_Empty exception is raised if this routine is called with an empty stack.

3.3.7.1.9.7.10.3.1 REQUIREMENTS ALLOCATION

This part meets CAMP requirement R166.

3.3.7.1.9.7.10.3.2 LOCAL ENTITIES DESIGN

None.

3.3.7.1.9.7.10.3.3 INPUT/OUTPUT

FORMAL PARAMETERS:

The following table describes this part's formal parameters:

Name	Type	Mode	Description
Stack	Stacks	in out	Bounded stack being manipulated
Old_Element	Elements	out	Element retrieved from the stack

3.3.7.1.9.7.10.3.4 LOCAL DATA

None.

3.3.7.1.9.7.10.3.5 PROCESS CONTROL

Not applicable.

3.3.7.1.9.7.10.3.6 PROCESSING

The following describes the processing performed by this part:

```
   procedure Retrieve_Element (Stack       : in out Stacks;
                               Old_Element :    out Elements) is

--    --------------------
--    --declaration section
--    --------------------

      List : Lists              renames Stack.List;
      Top  : Stack_Length_Range renames Stack.Top;

-- ---------------------------------
-- --begin procedure Retrieve_Element
-- ---------------------------------

   begin

--    --make sure there is something in the stack to retrieve
      if Top = 0 then
         raise Stack_Empty;
      end if;

--    --retrieve and remove the top element from the stack
      Old_Element := List(Top);
      Top         := Top - 1;

   end Retrieve_Element ;
```

3.3.7.1.9.7.10.3.7 UTILIZATION OF OTHER ELEMENTS

UTILIZATION OF OTHER ELEMENTS IN TOP LEVEL COMPONENT:

The following tables describe the elements used by this part but defined elsewhere in the parent top level component:

Data types:

The following table summarizes the types required by this part and defined as generic formal parameters to the Abstract_Data_Structures.Bounded_Stack package:

Name	Type	Description
Elements	private	User defined type of data contained in the stack

The following table summarizes the types required by this part and defined in the package specification of the Abstract_Data_Structures.Bounded_Stack package:

Name	Type	Range	Description
Stack_ Length_ Range	POSITIVE subtype	1 .. Stack_Size	Used to dimension the list of elements

The following data types were previously defined in the private part of this part's package specification:

Name	Type	Range	Description
Lists	array	N/A	List of elements
Stacks	record	N/A	List of data along with relevant information

Data objects:

The following table summarizes the objects required by this part and defined in the package specification of Abstract_Data_Structures. Bounded_Stack package:

Name	Type	Value	Description
Stack_ Size	POSITIVE	Initial_ Stack_Size	Number of elements in the stack

Exceptions:

The following table summarizes the exceptions required by this part and defined in the package specification of Abstract_Data_Structures. Bounded_Stack package:

Name	Description
Stack_Empty	Error condition raised if an attempt is made to look at or retrieve elements from an empty stack

3.3.7.1.9.7.10.3.8 LIMITATIONS

The following table describes the exceptions raised by this part:

Name	When/Why Raised
Stack_Empty	Raised if an attempt is made to look at or retrieve elements from an empty stack

3.3.7.1.9.7.10.4 PEEK UNIT DESIGN

This function returns the data in the top element of the stack, but does not remove the element from the stack.

A Stack_Empty exception is raised if this routine is called with an empty stack.

3.3.7.1.9.7.10.4.1 REQUIREMENTS ALLOCATION

This part meets CAMP requirement R166.

3.3.7.1.9.7.10.4.2 LOCAL ENTITIES DESIGN

None.

3.3.7.1.9.7.10.4.3 INPUT/OUTPUT

FORMAL PARAMETERS:

The following table describes this part's formal parameters:

Name	Type	Mode	Description
Stack	Stacks	in out	Bounded stack being manipulated

3.3.7.1.9.7.10.4.4 LOCAL DATA

None.

3.3.7.1.9.7.10.4.5 PROCESS CONTROL

Not applicable.

3.3.7.1.9.7.10.4.6 PROCESSING

The following describes the processing performed by this part:

```
   function Peek (Stack : in Stacks) return Elements is

--    ---------------------
--    --declaration section
--    ---------------------

      List : Lists              renames Stack.List;
      Top  : Stack_Length_Range renames Stack.Top;

-- ---------------------
-- --begin function Peek
-- ---------------------

   begin

--    --make sure there is something in the stack
      if Top = 0 then
         raise Stack_Empty;
      end if;

--    --return value in top element of the stack
      return List(Top);

   end Peek ;
```

3.3.7.1.9.7.10.4.7 UTILIZATION OF OTHER ELEMENTS

UTILIZATION OF OTHER ELEMENTS IN TOP LEVEL COMPONENT:

The following tables describe the elements used by this part but defined elsewhere in the parent top level component:

Data types:

The following table summarizes the types required by this part and defined as generic formal parameters to the Abstract_Data_Structures.Bounded_Stack package:

Name	Type	Description
Elements	private	User defined type of data contained in the stack

The following table summarizes the types required by this part and defined in the package specification of the Abstract_Data_Structures.Bounded_Stack package:

Name	Type	Range	Description
Stack_Length_Range	POSITIVE subtype	1 .. Stack_Size	Used to dimension the list of elements

The following data types were previously defined in the private part of this part's package specification:

Name	Type	Range	Description
Lists	array	N/A	List of elements
Stacks	record	N/A	List of data along with relevant information

Data objects:

The following table summarizes the objects required by this part and defined in the package specification of Abstract_Data_Structures. Bounded_Stack package:

Name	Type	Value	Description
Stack_Size	POSITIVE	Initial_Stack_Size	Number of elements in the stack

Exceptions:

The following table summarizes the exceptions required by this part and defined in the package specification of Abstract_Data_Structures. Bounded_Stack package:

Name	Description
Stack_Empty	Error condition raised if an attempt is made to look at or retrieve elements from an empty stack

3.3.7.1.9.7.10.4.8 LIMITATIONS

The following table describes the exceptions raised by this part:

Name	When/Why Raised
Stack_Empty	Raised if an attempt is made to look at or retrieve elements from an empty stack

3.3.7.1.9.7.10.5 STACK_STATUS UNIT DESIGN

This function returns the status of the stack based on the following algorithm:

if no elements are in the stack then stack status is empty elsif the maximum number of elements are in the stack then stack status is full else stack status is available end if

3.3.7.1.9.7.10.5.1 REQUIREMENTS ALLOCATION

This part meets CAMP requirement R166.

3.3.7.1.9.7.10.5.2 LOCAL ENTITIES DESIGN

None.

3.3.7.1.9.7.10.5.3 INPUT/OUTPUT

FORMAL PARAMETERS:

The following table describes this part's formal parameters:

Name	Type	Mode	Description
Stack	Stacks	in out	Bounded stack being manipulated

3.3.7.1.9.7.10.5.4 LOCAL DATA

Data objects:

The following table describes the data objects maintained by this part:

Name	Type	Value	Description
Status	Stack_Statuses	N/A	Status of the stack

3.3.7.1.9.7.10.5.5 PROCESS CONTROL

Not applicable.

3.3.7.1.9.7.10.5.6 PROCESSING

The following describes the processing performed by this part:

```
   function Stack_Status (Stack : in Stacks) return Stack_Statuses is

--    ---------------------
--    --declaration section
--    ---------------------

      Top    : Stack_Length_Range renames Stack.Top;

      Status : Stack_Statuses;

-- ----------------------------
-- --begin function Stack_Status
-- ----------------------------

   begin

      if Top = 0 then
         Status := Empty;
      elsif Top = Stack_Size then
         Status := Full;
      else
         Status := Available;
      end if;

      return Status;

   end Stack_Status ;
```

3.3.7.1.9.7.10.5.7 UTILIZATION OF OTHER ELEMENTS

UTILIZATION OF OTHER ELEMENTS IN TOP LEVEL COMPONENT:

The following tables describe the elements used by this part but defined elsewhere in the parent top level component:

Data types:

The following table summarizes the types required by this part and defined in the package specification of the Abstract_Data_Structures.Bounded_Stack package:

Name	Type	Range	Description
Stack_Length_Range	POSITIVE subtype	1 .. Stack_Size	Used to dimension the list of elements
Stack_Statuses	discrete type	Empty, Available, Full	Used to indicate the status of the stack

The following data types were previously defined in the private part of this part's package specification:

Name	Type	Range	Description
Stacks	record	N/A	List of data along with relevant information

Data objects:

The following table summarizes the objects required by this part and defined in the package specification of Abstract_Data_Structures. Bounded_Stack package:

Name	Type	Value	Description
Stack_Size	POSITIVE	Initial_Stack_Size	Number of elements in the stack

3.3.7.1.9.7.10.5.8 LIMITATIONS

None.

3.3.7.1.9.7.10.6 STACK_LENGTH UNIT DESIGN

This function returns the length of the stack.

3.3.7.1.9.7.10.6.1 REQUIREMENTS ALLOCATION

This part meets CAMP requirement R166.

3.3.7.1.9.7.10.6.2 LOCAL ENTITIES DESIGN

None.

3.3.7.1.9.7.10.6.3 INPUT/OUTPUT

FORMAL PARAMETERS:

The following table describes this part's formal parameters:

Name	Type	Mode	Description
Stack	Stacks	in out	Bounded stack being manipulated

3.3.7.1.9.7.10.6.4 LOCAL DATA

None.

3.3.7.1.9.7.10.6.5 PROCESS CONTROL

Not applicable.

3.3.7.1.9.7.10.6.6 PROCESSING

The following describes the processing performed by this part:

```
function Stack_Length (Stack : in Stacks) return Stack_Length_Range is
begin
   return Stack.Top;
end Stack_Length ;
```

3.3.7.1.9.7.10.6.7 UTILIZATION OF OTHER ELEMENTS

UTILIZATION OF OTHER ELEMENTS IN TOP LEVEL COMPONENT:

The following tables describe the elements used by this part but defined elsewhere in the parent top level component:

Data types:

The following table summarizes the types required by this part and defined in the package specification of the Abstract_Data_Structures.Bounded_Stack package:

Name	Type	Range	Description
Stack_Length_Range	POSITIVE subtype	1 .. Stack_Size	Used to dimension the list of elements

The following data types were previously defined in the private part of this part's package specification:

Name	Type	Range	Description
Stacks	record	N/A	List of data along with relevant information

3.3.7.1.9.7.10.6.8 LIMITATIONS

None.

3.3.7.1.9.8 UNBOUNDED_STACK PACKAGE DESIGN (CATALOG #P336-0)

This generic package performs last-in-first-out stacking operations on incoming data. The head of the stack always points to the last element added to the stack and the next element to be removed. The tail always points to a dummy node located below the oldest element on the stack. If head and tail point to the same node, the stack is empty.

An available space list is maintained local to this part. When this part is elaborated the available space list will have a dummy node plus Initial_Available_Space_Size nodes. When nodes are added to the stack, the Add_Element routine will try to get a node from the available space list before attempting to allocate more memory. When the Retrieve_Element routine is called, the unused node will be returned to the available space list for later use. The memory committed to the available space may be deallocated by calling the Free_Memory procedure.

The decomposition for this part is the same as that shown in the Top-Level Design Document.

3.3.7.1.9.8.1 REQUIREMENTS ALLOCATION

This part meets CAMP requirement R167.

3.3.7.1.9.8.2 LOCAL ENTITIES DESIGN

Data structures:

This part maintains an available space list local to the package body.

Subprograms:

The following subprograms are contained local to this body:

Name	Type	Description
Free_Node	procedure	Instantiation of UNCHECKED_DEALLOCATION
Dot_Next	function	Given a pointer P, this function returns the value of P.Next
Set_Next	procedure	Given two points P & Q, this procedure sets P.Next = Q

The following subprograms are contained in this part as a result of renaming operations on identically named routines contained in the locally instantiated Available_Space_Operations package.

Name	Type	Description
New_Node	function	Returns a node to the calling routine; will get a node from the available space list if possible, otherwise will allocate a new node
Save_Node	procedure	Handles placing a node in the available space list
Save_List	procedure	Handles placing a list of nodes in the available space list

This package body contains code to initialize the Available_Space_List. This code is executed when the package is elaborated. At a minimum, this code calls the Initialize procedure to initialize the Available_Space_List so it contains a dummy node. If the generic formal object Initial_Available_Space_Size is greater than or equal to 1, this routine then places the requested number of nodes (in addition to the dummy node) in the available space list.

3.3.7.1.9.8.3 INPUT/OUTPUT

GENERIC PARAMETERS:

The following generic parameters were previously defined when this part was specified in the package specification of the Abstract_Data_Structures package:

Data types:

The following table summarizes the generic formal types required by this part:

Name	Type	Description
Elements	private	User defined type of data contained in the stack

Data objects:

The following table summarizes the generic formal objects required by this part:

Name	Type	Description
Initial_Available_Space_Size	NATURAL	Number of nodes to be initially placed in the available space list

3.3.7.1.9.8.4 LOCAL DATA

None.

3.3.7.1.9.8.5 PROCESS CONTROL

Not applicable.

3.3.7.1.9.8.6 PROCESSING

The following describes the processing performed by this part:

```
with UNCHECKED_DEALLOCATION;
separate (Abstract_Data_Structures)
package body Unbounded_Stack is

-- ---------------------
-- --declaration section
-- ---------------------

-- --this pointer is accessed ONLY when setting up the Available_Space
   Initial_Head : Pointers := new Nodes;

   Available_Space : Stacks := (Current_Length => 0,
                                Top            => Initial_Head,
                                Bottom         => Initial_Head);

   Available_Length : INTEGER  renames Available_Space.Current_Length;
   Available_Top    : Pointers renames Available_Space.Top;
   Available_Bottom : Pointers renames Available_Space.Bottom;

   procedure Free is new UNCHECKED_DEALLOCATION
                           (Object => Nodes,
                            Name   => Pointers);

   procedure Free_Node (Which_Node : in out Pointers)
             renames Free;

   function Dot_Next (Ptr : in Pointers) return Pointers;

   procedure Set_Next (Ptr          : in Pointers;
                       Ptr_dot_Next : in Pointers);

   package Available_Space_Operations is new
           Available_Space_List_Operations
              (Nodes                    => Nodes,
               Pointers                 => Pointers,
```

```
                Available_Length        => Available_Length,
                Available_Head          => Available_Top,
                Available_Tail          => Available_Bottom);

   function New_Node return Pointers
            renames Available_Space_Operations.New_Node;

   procedure Save_Node (Saved_Node : in Pointers)
            renames Available_Space_Operations.Save_Node;

   procedure Save_List (Saved_Head : in Pointers;
                        Saved_Tail : in Pointers;
                        Node_Count : in POSITIVE)
            renames Available_Space_Operations.Save_List;


------------------------------------------
--begin package Unbounded_Stack
--(see header for package body for details)
------------------------------------------
begin

-- --set up available space list if one is desired
   if Initial_Available_Space_Size > 0 then

      Add_Nodes_to_Available_Space_List:
         for I in 1..Initial_Available_Space_Size loop
            Available_Bottom.Next := NEW Nodes;
            Available_Bottom      := Available_Bottom.Next;
         end loop Add_Nodes_to_Available_Space_List;

      Available_Length := Initial_Available_Space_Size;

   end if;

end Unbounded_Stack;
```

3.3.7.1.9.8.7 UTILIZATION OF OTHER ELEMENTS

Subprograms and task entries:

The following table describes the subroutines required by this part:

Name	Type	Source	Description
UNCHECKED_DEALLOCATION	generic function	N/A	Used to deallocate memory

Exceptions:

The following table describes the exceptions required by this part and defined in the Ada predefined package STANDARD:

Name	Description
STORAGE_ERROR	Raised when an attempt is made to dynamically allocate more memory than is available

UTILIZATION OF OTHER ELEMENTS IN TOP LEVEL COMPONENT:

The following tables describe the elements used by this part but defined elsewhere in the parent top level component:

Packages:

The following table describes the packages required by this part and specified in the package body of the Abstract_Data_Structures package:

Name	Type	Description
Available_Space_List_Operations	generic package	Contains the routines required to retrieve a node from and place a node in the available space list

Data types:

The following data types were previously defined in this part's package specification:

Name	Type	Range	Description
Stack_Statuses	discrete type	Empty, Available, Uninitialized	Indicates the current status of the stack

The following table describes the data types defined in the private part of the Abstract_Data_Structures.Unbounded_Stack package:

Name	Type	Range	Description
Nodes	record	N/A	Contains a single element and a pointer to another node
Pointers	access	N/A	Points to a node
Stacks	record	N/A	List of data along with relevant information

Exceptions:

The following exceptions were previously defined in this part's package specification:

Name	Description
Stack_Empty	Error condition raised if an attempt is made to look at or retrieve elements from an empty stack
Stack_Not_Initialized	Raised if an attempt is made to use an uninitialized stack

3.3.7.1.9.8.8 LIMITATIONS

The following table describes the exceptions raised by this part:

Name	When/Why Raised
STANDARD.STORAGE_ERROR	Raised during elaboration of this package if an attempt is made to allocate memory when no more is available

3.3.7.1.9.8.9 LLCSC DESIGN

None.

3.3.7.1.9.8.10 UNIT DESIGN

3.3.7.1.9.8.10.1 INITIALIZE UNIT DESIGN

This procedure initializes a stack by placing a dummy node in the stack, pointing the top and bottom to the dummy node, and setting the length to 0. If this routine is called with a stack containing elements, then the stack is cleared of all but the dummy node.

3.3.7.1.9.8.10.1.1 REQUIREMENTS ALLOCATION

This part meets CAMP requirement R167.

3.3.7.1.9.8.10.1.2 LOCAL ENTITIES DESIGN

None.

3.3.7.1.9.8.10.1.3 INPUT/OUTPUT

FORMAL PARAMETERS:

The following table describes this part's formal parameters:

Name	Type	Mode	Description
Stack	Stacks	in out	Stack being manipulated

3.3.7.1.9.8.10.1.4 LOCAL DATA

None.

3.3.7.1.9.8.10.1.5 PROCESS CONTROL

Not applicable.

3.3.7.1.9.8.10.1.6 PROCESSING

The following describes the processing performed by this part:

```
   procedure Initialize (Stack : in out Stacks) is

--    ---------------------
--    --declaration section
--    ---------------------

      Current_Length : INTEGER  renames Stack.Current_Length;
      Top            : Pointers renames Stack.Top;
      Bottom         : Pointers renames Stack.Bottom;

-- ----------------------------
-- --begin procedure Initialize
-- ----------------------------

   begin

      if Current_Length = -1 then

--       --handle an uninitialized stack
         Top            := New_Node;
         Bottom         := Top;
         Current_Length := 0;

      elsif Current_Length > 0 then

--       --handle a stack that has elements in it
         Clear_Stack (Stack => Stack);

      else

--       --current length = 0, so do nothing
         NULL;

      end if;

   end Initialize ;
```

3.3.7.1.9.8.10.1.7 UTILIZATION OF OTHER ELEMENTS

UTILIZATION OF OTHER ELEMENTS IN TOP LEVEL COMPONENT:

The following tables describe the elements used by this part but defined elsewhere in the parent top level component:

Subprograms and task entries:

The following table summarizes the subroutines required by this part and defined in the package specification of Abstract_Data_Structures.Unbounded_Stack:

Name	Type	Description
Clear_Stack	procedure	Clears a stack by returning all of its nodes to the available space list

The following table summarizes the subroutines and task entries required by this part and defined in the package body of Abstract_Data_Structures.Unbounded_Stack:

Name	Type	Description
New_Node	function	Returns a node to the calling routine; will get a node from the available space list if possible, otherwise will allocate a new node

Data types:

The following table describes the data types defined in the private part of the Abstract_Data_Structures.Unbounded_Stack package:

Name	Type	Range	Description
Nodes	record	N/A	Contains a single element and a pointer to another node
Pointers	access	N/A	Points to a node
Stacks	record	N/A	List of data along with relevant information

3.3.7.1.9.8.10.1.8 LIMITATIONS

The following table describes the exceptions raised by this part:

Name	When/Why Raised
STANDARD.STORAGE_ERROR	Raised if an attempt is made to allocate more memory than is available

3.3.7.1.9.8.10.2 CLEAR_STACK UNIT DESIGN

This procedure removes nodes from a stack, leaving only the dummy node. The nodes removed are placed in the available space list.

A Stack_Not_Initialized exception is raised if this routine is called with an uninitialized stack.

3.3.7.1.9.8.10.2.1 REQUIREMENTS ALLOCATION

This part meets CAMP requirement R167.

3.3.7.1.9.8.10.2.2 LOCAL ENTITIES DESIGN

None.

3.3.7.1.9.8.10.2.3 INPUT/OUTPUT

FORMAL PARAMETERS:

The following table describes this part's formal parameters:

Name	Type	Mode	Description
Stack	Stacks	in out	Stack being manipulated

3.3.7.1.9.8.10.2.4 LOCAL DATA

Data objects:

The following table describes the data objects maintained by this part:

Name	Type	Value	Description
This_Node	Pointers	N/A	Points to the node to be placed in the available space list

3.3.7.1.9.8.10.2.5 PROCESS CONTROL

Not applicable.

3.3.7.1.9.8.10.2.6 PROCESSING

The following describes the processing performed by this part:

```
   procedure Clear_Stack (Stack : in out Stacks) is

--    ---------------------
--    --declaration section
--    ---------------------

      Current_Length : INTEGER  renames Stack.Current_Length;
      Top            : Pointers renames Stack.Top;
      Bottom         : Pointers renames Stack.Bottom;

      This_Node      : Pointers;

-- ----------------------------
-- --begin procedure Clear_Stack
-- ----------------------------

   begin

--    --make sure stack has been initialized
      if Current_Length = -1 then

         raise Stack_Not_Initialized;

--    --make sure there is something in the stack
      elsif Current_Length /= 0 then

--       --placed nodes in the available space list
         Save_List (Saved_Head => Top.Next,
                    Saved_Tail => Bottom,
                    Node_Count => Current_Length);

--       --reinitialize stack variables
         Top.Next       := NULL;
         Bottom         := Top;
         Current_Length := 0;

      end if;

   end Clear_Stack ;
```

3.3.7.1.9.8.10.2.7 UTILIZATION OF OTHER ELEMENTS

UTILIZATION OF OTHER ELEMENTS IN TOP LEVEL COMPONENT:

The following tables describe the elements used by this part but defined elsewhere in the parent top level component:

Subprograms and task entries:

The following table summarizes the subroutines and task entries required by this part and defined in the package body of Abstract_Data_-Structures.Unbounded_Stack:

Name	Type	Description
Save_List	procedure	Handles placing a list of nodes in the available space list

Data types:

The following table describes the data types defined in the private part of the Abstract_Data_Structures.Unbounded_Stack package:

Name	Type	Range	Description
Nodes	record	N/A	Contains a single element and a pointer to another node
Pointers	access	N/A	Points to a node
Stacks	record	N/A	List of data along with relevant information

Exceptions:

The following table summarizes the exceptions required by this part and defined in the package specification of Abstract_Data_Structures. Unbounded_Stack:

Name	Description
Stack_Not_Initialized	Raised if an attempt is made to use an uninitialized stack

3.3.7.1.9.8.10.2.8 LIMITATIONS

The following table describes the exceptions raised by this part:

Name	When/Why Raised
Stack_Not_Initialized	Raised if an attempt is made to manipulate an uninitialized stack

3.3.7.1.9.8.10.3 FREE_MEMORY UNIT DESIGN

This procedure deallocates the memory occupied by the nodes in the available space list. Only a dummy node will be left in the list.

3.3.7.1.9.8.10.3.1 REQUIREMENTS ALLOCATION

This part meets CAMP requirement R167.

3.3.7.1.9.8.10.3.2 LOCAL ENTITIES DESIGN

None.

3.3.7.1.9.8.10.3.3 INPUT/OUTPUT

None.

3.3.7.1.9.8.10.3.4 LOCAL DATA

Data objects:

The following table describes the data objects maintained by this part:

Name	Type	Value	Description
This_Node	Pointers	N/A	Points to the node to be deallocated

3.3.7.1.9.8.10.3.5 PROCESS CONTROL

Not applicable.

3.3.7.1.9.8.10.3.6 PROCESSING

The following describes the processing performed by this part:

```
   procedure Free_Memory is

--    ---------------------
--    --declaration section
--    ---------------------

      This_Node : Pointers;

-- ----------------------------
-- --begin procedure Free_Memory
-- ----------------------------

   begin
```

```
      Deallocate_Nodes_in_Available_Space_List:
         while Available_Top /= Available_Bottom loop

            This_Node := Available_Top;
            Available_Top := Available_Top.Next;
            Free_Node (Which_Node => This_Node);

         end loop Deallocate_Nodes_in_Available_Space_List;

      Available_Length   := 0;
      Available_Top.Next := NULL;

   end Free_Memory ;
```

3.3.7.1.9.8.10.3.7 UTILIZATION OF OTHER ELEMENTS

UTILIZATION OF OTHER ELEMENTS IN TOP LEVEL COMPONENT:

The following tables describe the elements used by this part but defined elsewhere in the parent top level component:

Subprograms and task entries:

The following table summarizes the subroutines and task entries required by this part and defined in the package body of Abstract_Data_-Structures.Unbounded_Stack:

Name	Type	Description
Free_Node	procedure	Instantiation of UNCHECKED_DEALLOCATION

Data types:

The following table describes the data types defined in the private part of the Abstract_Data_Structures.Unbounded_Stack package:

Name	Type	Range	Description
Nodes	record	N/A	Contains a single element and a pointer to another node
Pointers	access	N/A	Points to a node
Stacks	record	N/A	List of data along with relevant information

3.3.7.1.9.8.10.3.8 LIMITATIONS

None.

3.3.7.1.9.8.10.4 ADD_ELEMENT UNIT DESIGN

This procedure adds an element to the top of the stack.

A Stack_Not_Initialized exception is raised if this routine is called with an uninitialized stack.

3.3.7.1.9.8.10.4.1 REQUIREMENTS ALLOCATION

This part meets CAMP requirement R167.

3.3.7.1.9.8.10.4.2 LOCAL ENTITIES DESIGN

None.

3.3.7.1.9.8.10.4.3 INPUT/OUTPUT

FORMAL PARAMETERS:

The following table describes this part's formal parameters:

Name	Type	Mode	Description
New_Element	Elements	in	Element to be added to the stack
Stack	Stacks	in out	Stack being manipulated

3.3.7.1.9.8.10.4.4 LOCAL DATA

Data objects:

The following table describes the data objects maintained by this part:

Name	Type	Value	Description
Ptr	Pointers	N/A	Points to the new node to be placed in the stack

3.3.7.1.9.8.10.4.5 PROCESS CONTROL

Not applicable.

3.3.7.1.9.8.10.4.6 PROCESSING

The following describes the processing performed by this part:

```
   procedure Add_Element (New_Element : in     Elements;
                          Stack       : in out Stacks) is
```

```
--    --------------------
--    --declaration section
--    --------------------

      Current_Length : INTEGER  renames Stack.Current_Length;
      Top            : Pointers renames Stack.Top;

      Ptr            : Pointers;

-- ----------------------------
-- --begin procedure Add_Element
-- ----------------------------

   begin

      if Current_Length = -1 then
         raise Stack_Not_Initialized;
      end if;

--    --get a node and initialize it
      Ptr      := New_Node;
      Ptr.Data := New_Element;

--    --place the node on the stack
      Ptr.Next       := Top;
      Top            := Ptr;
      Current_Length := Current_Length + 1;

   end Add_Element ;
```

3.3.7.1.9.8.10.4.7 UTILIZATION OF OTHER ELEMENTS

UTILIZATION OF OTHER ELEMENTS IN TOP LEVEL COMPONENT:

The following tables describe the elements used by this part but defined elsewhere in the parent top level component:

Subprograms and task entries:

The following table summarizes the subroutines and task entries required by this part and defined in the package body of Abstract_Data_-Structures.Unbounded_Stack:

Name	Type	Description
New_Node	function	Returns a node to the calling routine; will get a node from the available space list if possible, otherwise will allocate a new node

Data types:

The following table summarizes the types required by this part and defined as generic formal parameters to the Abstract_Data_Structures. Unbounded_Stack package:

Name	Type	Description
Elements	private	User defined type of data contained in the stack

The following table describes the data types defined in the private part of the Abstract_Data_Structures.Unbounded_Stack package:

Name	Type	Range	Description
Nodes	record	N/A	Contains a single element and a pointer to another node
Pointers	access	N/A	Points to a node
Stacks	record	N/A	List of data along with relevant information

Exceptions:

The following table summarizes the exceptions required by this part and defined in the package specification of Abstract_Data_Structures. Unbounded_Stack:

Name	Description
Stack_Not_Initialized	Raised if an attempt is made to use an uninitialized stack

3.3.7.1.9.8.10.4.8 LIMITATIONS

The following table describes the exceptions raised by this part:

Name	When/Why Raised
STANDARD.STORAGE_ERROR	Raised if an attempt is made to allocate more memory than is available
Stack_Not_Initialized	Raised if an attempt is made to manipulate an uninitialized stack

3.3.7.1.9.8.10.5 RETRIEVE_ELEMENT UNIT DESIGN

This procedure retrieves the top element of the stack and returns the data in it to the calling routine. The node is then placed in the available space list.

A Stack_Empty exception is raised if this routine is called with an empty stack.

A Stack_Not_Initialized exception is raised if this routine is called with an uninitialized stack.

3.3.7.1.9.8.10.5.1 REQUIREMENTS ALLOCATION

This part meets CAMP requirement R167.

3.3.7.1.9.8.10.5.2 LOCAL ENTITIES DESIGN

None.

3.3.7.1.9.8.10.5.3 INPUT/OUTPUT

FORMAL PARAMETERS:

The following table describes this part's formal parameters:

Name	Type	Mode	Description
Stack	Stacks	in out	Stack being manipulated
Old_Element	Elements	out	Elements retrieved from the stack

3.3.7.1.9.8.10.5.4 LOCAL DATA

Data objects:

The following table describes the data objects maintained by this part:

Name	Type	Value	Description
This_Node	Pointers	N/A	Node to be returned to the available space list

3.3.7.1.9.8.10.5.5 PROCESS CONTROL

Not applicable.

3.3.7.1.9.8.10.5.6 PROCESSING

The following describes the processing performed by this part:

```
   procedure Retrieve_Element (Stack       : in out Stacks;
                               Old_Element :    out Elements) is
```

```
--    --------------------
--    --declaration section
--    --------------------

      Current_Length : INTEGER  renames Stack.Current_Length;
      Top            : Pointers renames Stack.Top;

      This_Node      : Pointers;

-- ---------------------------------
-- --begin procedure Retrieve_Element
-- ---------------------------------

   begin

--    --make sure there is something to retrieve
      if Current_Length = -1 then
         raise Stack_Not_Initialized;
      elsif Current_Length = 0 then
         raise Stack_Empty;
      end if;

--    --retrieve data in the top node
      Old_Element := Top.Data;

--    --dispose of top node and adjust the stack
      This_Node := Top;
      Top       := Top.Next;
      Save_Node (Saved_Node => This_Node);
      Current_Length := Current_Length - 1;

   end Retrieve_Element ;
```

3.3.7.1.9.8.10.5.7 UTILIZATION OF OTHER ELEMENTS

UTILIZATION OF OTHER ELEMENTS IN TOP LEVEL COMPONENT:

The following tables describe the elements used by this part but defined elsewhere in the parent top level component:

Subprograms and task entries:

The following table summarizes the subroutines and task entries required by this part and defined in the package body of Abstract_Data_-Structures.Unbounded_Stack:

Name	Type	Description
Save_Node	procedure	Handles placing a node in the available space list

Data types:

The following table summarizes the types required by this part and defined as generic formal parameters to the Abstract_Data_Structures. Unbounded_Stack package:

Name	Type	Description
Elements	private	User defined type of data contained in the stack

The following table describes the data types defined in the private part of the Abstract_Data_Structures.Unbounded_Stack package:

Name	Type	Range	Description
Nodes	record	N/A	Contains a single element and a pointer to another node
Pointers	access	N/A	Points to a node
Stacks	record	N/A	List of data along with relevant information

Exceptions:

The following table summarizes the exceptions required by this part and defined in the package specification of Abstract_Data_Structures. Unbounded_Stack:

Name	Description
Stack_Empty	Error condition raised if an attempt is made to look at or retrieve elements from an empty stack
Stack_Not_Initialized	Raised if an attempt is made to use an uninitialized stack

3.3.7.1.9.8.10.5.8 LIMITATIONS

The following table describes the exceptions raised by this part:

Name	When/Why Raised
Stack_Not_Initialized	Raised if an attempt is made to manipulate an uninitialized stack
Stack_Empty	Raised if an attempt is made to retrieve or look at elements in an empty stack

3.3.7.1.9.8.10.6 PEEK UNIT DESIGN

This function returns the data contained in the top element of the stack, but does not remove the element from the stack.

A Stack_Empty exception is raised if this routine is called with an empty stack.

A Stack_Not_Initialized exception is raised if this routine is called with an uninitialized stack.

3.3.7.1.9.8.10.6.1 REQUIREMENTS ALLOCATION

This part meets CAMP requirement R167.

3.3.7.1.9.8.10.6.2 LOCAL ENTITIES DESIGN

None.

3.3.7.1.9.8.10.6.3 INPUT/OUTPUT

FORMAL PARAMETERS:

The following table describes this part's formal parameters:

Name	Type	Mode	Description
Stack	Stacks	in out	Stack being manipulated

3.3.7.1.9.8.10.6.4 LOCAL DATA

None.

3.3.7.1.9.8.10.6.5 PROCESS CONTROL

Not applicable.

3.3.7.1.9.8.10.6.6 PROCESSING

The following describes the processing performed by this part:

```
   function Peek (Stack : in Stacks) return Elements is

--    ---------------------
--    --declaration section
--    ---------------------

      Current_Length : INTEGER  renames Stack.Current_Length;
      Top            : Pointers renames Stack.Top;

-- ---------------------
-- --begin function Peek
-- ---------------------
```

```
   begin

--     --make sure there is something to peek at
       if Current_Length = -1 then
          raise Stack_Not_Initialized;
       elsif Current_Length = 0 then
          raise Stack_Empty;
       end if;

--     --returned desired element
       return Top.Data;

   end Peek ;
```

3.3.7.1.9.8.10.6.7 UTILIZATION OF OTHER ELEMENTS

UTILIZATION OF OTHER ELEMENTS IN TOP LEVEL COMPONENT:

The following tables describe the elements used by this part but defined elsewhere in the parent top level component:

Data types:

The following table summarizes the types required by this part and defined as generic formal parameters to the Abstract_Data_Structures. Unbounded_Stack package:

Name	Type	Description
Elements	private	User defined type of data contained in the stack

The following table describes the data types defined in the private part of the Abstract_Data_Structures.Unbounded_Stack package:

Name	Type	Range	Description
Nodes	record	N/A	Contains a single element and a pointer to another node
Pointers	access	N/A	Points to a node
Stacks	record	N/A	List of data along with relevant information

Exceptions:

The following table summarizes the exceptions required by this part and defined in the package specification of Abstract_Data_Structures. Unbounded_Stack:

Name	Description
Stack_Empty	Error condition raised if an attempt is made to look at or retrieve elements from an empty stack
Stack_Not_Initialized	Raised if an attempt is made to use an uninitialized stack

3.3.7.1.9.8.10.6.8 LIMITATIONS

The following table describes the exceptions raised by this part:

Name	When/Why Raised
Stack_Not_Initialized	Raised if an attempt is made to manipulate an uninitialized stack
Stack_Empty	Raised if an attempt is made to retrieve or look at elements in an empty stack

3.3.7.1.9.8.10.7 STACK_STATUS UNIT DESIGN

This function returns the status of the stack according to the following algorithm:

if stack has never been initialized then stack status is uninitialized elsif stack has no elements in it then stack status in empty else stack status is available end if

3.3.7.1.9.8.10.7.1 REQUIREMENTS ALLOCATION

This part meets CAMP requirement R167.

3.3.7.1.9.8.10.7.2 LOCAL ENTITIES DESIGN

None.

3.3.7.1.9.8.10.7.3 INPUT/OUTPUT

FORMAL PARAMETERS:

The following table describes this part's formal parameters:

Name	Type	Mode	Description
Stack	Stacks	in out	Stack being manipulated

3.3.7.1.9.8.10.7.4 LOCAL DATA

Data objects:

The following table describes the data objects maintained by this part:

Name	Type	Value	Description
Status	Stack_Statuses	N/A	Status of the stack

3.3.7.1.9.8.10.7.5 PROCESS CONTROL

Not applicable.

3.3.7.1.9.8.10.7.6 PROCESSING

The following describes the processing performed by this part:

```
   function Stack_Status (Stack : in Stacks) return Stack_Statuses is

--    ---------------------
--    --declaration section
--    ---------------------

      Current_Length : INTEGER  renames Stack.Current_Length;

      Status         : Stack_Statuses;

-- -----------------------------
-- --begin function Stack_Status
-- -----------------------------

   begin

      if Current_Length = -1 then
         Status := Uninitialized;

      elsif Current_Length = 0 then
         Status := Empty;

      else
         Status := Available;

      end if;

      return Status;

   end Stack_Status ;
```

3.3.7.1.9.8.10.7.7 UTILIZATION OF OTHER ELEMENTS

UTILIZATION OF OTHER ELEMENTS IN TOP LEVEL COMPONENT:

The following tables describe the elements used by this part but defined elsewhere in the parent top level component:

Data types:

The following table summarizes the types required by this part and defined in the package specification of Abstract_Data_Structures. Unbounded_Stack:

Name	Type	Range	Description
Stack_ Statuses	discrete type	Empty, Available Uninitialized	Indicates the current status of the stack

The following table describes the data types defined in the private part of the Abstract_Data_Structures.Unbounded_Stack package:

Name	Type	Range	Description
Stacks	record	N/A	List of data along with relevant information

3.3.7.1.9.8.10.7.8 LIMITATIONS

None.

3.3.7.1.9.8.10.8 STACK_LENGTH UNIT DESIGN

This function returns the length of the stack.

A Stack_Not_Initialized exception is raised if this routine is called with an uninitialized stack.

3.3.7.1.9.8.10.8.1 REQUIREMENTS ALLOCATION

This part meets CAMP requirement R167.

3.3.7.1.9.8.10.8.2 LOCAL ENTITIES DESIGN

None.

3.3.7.1.9.8.10.8.3 INPUT/OUTPUT

FORMAL PARAMETERS:

The following table describes this part's formal parameters:

Name	Type	Mode	Description
Stack	Stacks	in out	Stack being manipulated

3.3.7.1.9.8.10.8.4 LOCAL DATA

None.

3.3.7.1.9.8.10.8.5 PROCESS CONTROL

Not applicable.

3.3.7.1.9.8.10.8.6 PROCESSING

The following describes the processing performed by this part:

```
   function Stack_Length (Stack : in Stacks) return NATURAL is

--    ---------------------
--    --declaration section
--    ---------------------

      Current_Length : INTEGER  renames Stack.Current_Length;

-- ----------------------------
-- --begin function Stack_Length
-- ----------------------------

   begin

--    --make sure stack has been initialized
      if Current_Length = -1 then
         raise Stack_Not_Initialized;
      end if;

      return Current_Length;

   end Stack_Length ;
```

3.3.7.1.9.8.10.8.7 UTILIZATION OF OTHER ELEMENTS

UTILIZATION OF OTHER ELEMENTS IN TOP LEVEL COMPONENT:

The following tables describe the elements used by this part but defined elsewhere in the parent top level component:

Data types:

The following table describes the data types defined in the private part of the Abstract_Data_Structures.Unbounded_Stack package:

Name	Type	Range	Description
Stacks	record	N/A	List of data along with relevant information

Exceptions:

The following table summarizes the exceptions required by this part and defined in the package specification of Abstract_Data_Structures. Unbounded_Stack:

Name	Description
Stack_Not_Initialized	Raised if an attempt is made to use an uninitialized stack

3.3.7.1.9.8.10.8.8 LIMITATIONS

The following table describes the exceptions raised by this part:

Name	When/Why Raised
Stack_Not_Initialized	Raised if an attempt is made to manipulate an uninitialized stack

3.3.7.1.9.8.10.9 DOT_NEXT UNIT DESIGN

Given an input pointer P, this function returns the value of P.Next.

3.3.7.1.9.8.10.9.1 REQUIREMENTS ALLOCATION

None.

3.3.7.1.9.8.10.9.2 LOCAL ENTITIES DESIGN

None.

3.3.7.1.9.8.10.9.3 INPUT/OUTPUT

FORMAL PARAMETERS:

The following table describes this part's formal parameters:

Name	Type	Mode	Description
Ptr	Pointers	in	Pointer to the node whose "next" entry is to be returned

3.3.7.1.9.8.10.9.4 LOCAL DATA

None.

3.3.7.1.9.8.10.9.5 PROCESS CONTROL

Not applicable.

3.3.7.1.9.8.10.9.6 PROCESSING

The following describes the processing performed by this part:

```
function Dot_Next (Ptr : in Pointers) return Pointers is
begin
   return Ptr.Next;
end Dot_Next;
```

3.3.7.1.9.8.10.9.7 UTILIZATION OF OTHER ELEMENTS

UTILIZATION OF OTHER ELEMENTS IN TOP-LEVEL COMPONENT:

The following tables describe the elements used by this part but defined elsewhere in the parent top-level component:

Data types:

The following table summarizes the types required by this part and defined in the private portion of the part's package specification:

Name	Type	Range	Description
Nodes	record	N/A	A single entity in the stack; contains data and a pointer to the next node
Pointers	access	N/A	Points to a node in the stack
Stacks	record	N/A	Record containing the value of the current length, head, and tail of the stack

3.3.7.1.9.8.10.9.8 LIMITATIONS

None.

3.3.7.1.9.8.10.10 SET_NEXT UNIT DESIGN

Given an two input pointers, P and Q, this procedure sets P.Next equal to Q.

3.3.7.1.9.8.10.10.1 REQUIREMENTS ALLOCATION

None.

3.3.7.1.9.8.10.10.2 LOCAL ENTITIES DESIGN

None.

3.3.7.1.9.8.10.10.3 INPUT/OUTPUT

FORMAL PARAMETERS:

The following table describes this part's formal parameters:

Name	Type	Mode	Description
Ptr	Pointers	in	Pointer to the node whose "next" entry is to be modified
Ptr_dot_Next	Pointers	in	Value to which Ptr.Next is to be set

3.3.7.1.9.8.10.10.4 LOCAL DATA

None.

3.3.7.1.9.8.10.10.5 PROCESS CONTROL

Not applicable.

3.3.7.1.9.8.10.10.6 PROCESSING

The following describes the processing performed by this part:

```
procedure Set_Next (Ptr          : in Pointers;
                    Ptr_dot_Next : in Pointers) is
begin
   Ptr.Next := Ptr_dot_Next;
end Set_Next;
```

3.3.7.1.9.8.10.10.7 UTILIZATION OF OTHER ELEMENTS

UTILIZATION OF OTHER ELEMENTS IN TOP-LEVEL COMPONENT:

The following tables describe the elements used by this part but defined elsewhere in the parent top-level component:

Data types:

The following table summarizes the types required by this part and defined in the private portion of the part's package specification:

Name	Type	Range	Description
Nodes	record	N/A	A single entity in the stack; contains data and a pointer to the next node
Pointers	access	N/A	Points to a node in the stack
Stacks	record	N/A	Record containing the value of the current length, head, and tail of the stack

3.3.7.1.9.8.10.10.8 LIMITATIONS

None.

3.3.7.1.10 UNIT DESIGN

None.

```
package body Abstract_Data_Structures is

pragma PAGE;
   generic
      type Nodes              is limited private;
      type Pointers           is access Nodes;
      Available_Length        : in out INTEGER;
      Available_Head          : in out Pointers;
      Available_Tail          : in out Pointers;
      with function  Dot_Next (Ptr : in Pointers) return Pointers is <>;
      with procedure Set_Next (Ptr          : in Pointers;
                               Ptr_Dot_Next : in Pointers) is <>;
   package Available_Space_List_Operations is

      function  New_Node return Pointers;

      procedure Save_Node (Saved_Node : in Pointers);

      procedure Save_List (Saved_Head : in Pointers;
                           Saved_Tail : in Pointers;
                           Node_Count : in POSITIVE);

   end Available_Space_List_Operations;

pragma PAGE;
-- -------------------------
-- --separate package bodies
-- -------------------------

   package body Bounded_Fifo_Buffer is separate;

   package body Unbounded_Fifo_Buffer is separate;

   package body Nonblocking_Circular_Buffer is separate;

   package body Unbounded_Priority_Queue is separate;

   package body Bounded_Stack is separate;

   package body Unbounded_Stack is separate;

   package body Available_Space_List_Operations is separate;

end Abstract_Data_Structures;
```

```
separate (Abstract_Data_Structures)
package body Available_Space_List_Operations is

pragma PAGE;
   function New_Node return Pointers is

--    ---------------------
--    --declaration section
--    ---------------------

      Ptr                : Pointers;
      New_Available_Head : Pointers;

-- -------------------------
-- --begin function New_Node
-- -------------------------

   begin

      if Available_Length > 0 then

--       --get the node from the available space list and mark the node
--       --that will now be the head of the available space list
         Ptr                := Available_Head;
         New_Available_Head := Dot_Next(Available_Head);

--       --initialize node being returned
         Set_Next (Ptr          => Ptr,
                   Ptr_Dot_Next => null);

--       --adjust the available space list
         Available_Head   := New_Available_Head;
         Available_Length := Available_Length - 1;

      else

--       --allocate space to get the node
         Ptr := new Nodes;

      end if;

      return Ptr;

   end New_Node;

pragma PAGE;
   procedure Save_Node(Saved_Node : in Pointers) is

   begin

      Set_Next (Ptr          => Available_Tail,
                Ptr_Dot_Next => Saved_Node);

      Available_Tail := Saved_Node;

      Set_Next (Ptr          => Available_Tail,
                Ptr_Dot_Next => null);
```

```
      Available_Length := Available_Length + 1;

   end Save_Node;

pragma PAGE;
   procedure Save_List (Saved_Head : in Pointers;
                        Saved_Tail : in Pointers;
                        Node_Count : in POSITIVE) is

   begin

      Set_Next (Ptr            => Available_Tail,
                Ptr_Dot_Next   => Saved_Head);

      Available_Tail := Saved_Tail;

      Set_Next (Ptr            => Available_Tail,
                Ptr_Dot_Next   => Saved_Head);

      Available_Length := Available_Length + Node_Count;

   end Save_List;

end Available_Space_List_Operations;
```

```
separate (Abstract_Data_Structures)
package body Bounded_Fifo_Buffer is

pragma PAGE;
   procedure Clear_Buffer (Buffer : out Buffers) is

--     ---------------------
--     --declaration section
--     ---------------------

       Buffer_Length : Buffer_Range renames Buffer.Buffer_Length;
       Head          : Buffer_Range renames Buffer.Head;
       Tail          : Buffer_Range renames Buffer.Tail;

-- ------------------------------
-- --begin procedure Clear_Buffer
-- ------------------------------

   begin

      Buffer_Length := 0;
      Head          := 0;
      Tail          := 1;

   end Clear_Buffer ;

pragma PAGE;
   procedure Add_Element (New_Element : in     Elements;
                          Buffer      : in out Buffers) is

--     ---------------------
--     --declaration section
--     ---------------------

       LIST          : Lists        renames Buffer.LIST;
       Buffer_Length : Buffer_Range renames Buffer.Buffer_Length;
       Head          : Buffer_Range renames Buffer.Head;
       Tail          : Buffer_Range renames Buffer.Tail;

-- ----------------------------
-- --begin procedure Add_Element
-- ----------------------------

   begin

--     --make sure buffer isn't full
       if Head = Tail then
          raise Buffer_Full;
       end if;

       LIST(Tail) := New_Element;
       Buffer_Length := Buffer_Length + 1;
       if Tail = Buffer_Size then
          Tail := 0;
       else
          Tail := Tail + 1;
       end if;
```

```
   end Add_Element ;

pragma PAGE;
   procedure Retrieve_Element (Buffer      : in out Buffers;
                               Old_Element :    out Elements) is

--     --------------------
--     --declaration section
--     --------------------

       Buffer_Length : Buffer_Range renames Buffer.Buffer_Length;
       Head          : Buffer_Range renames Buffer.Head;
       LIST          : Lists        renames Buffer.LIST;
       Tail          : Buffer_Range renames Buffer.Tail;

-- ---------------------------------
-- --begin procedure Retrieve_Element
-- ---------------------------------

   begin

--     --make sure don't have an empty buffer
       if Head = (Tail-1) or else (Tail = 0 and Head = Buffer_Size) then
          raise Buffer_Empty;
       end if;

       if Head = Buffer_Size then
          Head := 0;
       else
          Head := Head + 1;
       end if;
       Old_Element   := LIST(Head);
       Buffer_Length := Buffer_Length - 1;

   end Retrieve_Element ;

pragma PAGE;
   function Peek (Buffer : in Buffers) return Elements is

--     --------------------
--     --declaration section
--     --------------------

       Buffer_Length : Buffer_Range renames Buffer.Buffer_Length;
       Head          : Buffer_Range renames Buffer.Head;
       Tail          : Buffer_Range renames Buffer.Tail;
       LIST          : Lists        renames Buffer.LIST;
       Spot          : Buffer_Range;

-- ----------------------
-- --begin function Peek
-- --------------------

   begin

--     --make sure don't have an empty buffer
```

```
      if Head = (Tail-1) or else (Tail = 0 and Head = Buffer_Size) then
         raise Buffer_Empty;
      end if;

      if Head = Buffer_Size then
         Spot := 0;
      else
         Spot := Head + 1;
      end if;

      return LIST(Spot);

   end Peek ;

pragma PAGE;
   function Buffer_Status (Buffer : in Buffers) return Buffer_Statuses is

--    ---------------------
--    --declaration section
--    ---------------------

      Head : Buffer_Range renames Buffer.Head;
      Tail : Buffer_Range renames Buffer.Tail;

      Status : Buffer_Statuses;

-- ------------------------------
-- --begin function Buffer_Status
-- ------------------------------

   begin

      if Head = (Tail-1) or else (Tail = 0 and Head = Buffer_Size) then
         Status := Empty;
      elsif Head = Tail then
         Status := Full;
      else
         Status := Available;
      end if;

      return Status;

   end Buffer_Status ;

pragma PAGE;
   function Buffer_Length (Buffer : in Buffers) return Buffer_Range is

   begin

      return Buffer.Buffer_Length;

   end Buffer_Length ;

end Bounded_Fifo_Buffer;
```

```
with Unchecked_Deallocation;
separate (Abstract_Data_Structures)
package body Unbounded_Fifo_Buffer is

-- ---------------------
-- --declaration section
-- ---------------------

-- --this variable is accessed ONLY when setting up the available space list
   Initial_Head     : Pointers := new Nodes;

   Available_Space : Buffers := (Current_Length => 0,
                                 Head           => Initial_Head,
                                 Tail           => Initial_Head);

   Available_Length : INTEGER  renames Available_Space.Current_Length;
   Available_Head   : Pointers renames Available_Space.Head;
   Available_Tail   : Pointers renames Available_Space.Tail;

   procedure Free is new Unchecked_Deallocation
                          (Object => Nodes,
                           NAME   => Pointers);

   procedure Free_Node (Which_Node : in out Pointers)
             renames Free;

   function Dot_Next (Ptr : in Pointers) return Pointers;

   procedure Set_Next (Ptr         : in Pointers;
                       Ptr_Dot_Next : in Pointers);

   package Available_Space_Operations is new
           Available_Space_List_Operations
              (Nodes                => Nodes,
               Pointers             => Pointers,
               Available_Length     => Available_Length,
               Available_Head       => Available_Head,
               Available_Tail       => Available_Tail);

   function New_Node return Pointers
            renames Available_Space_Operations.New_Node;

   procedure Save_Node (Saved_Node : in Pointers)
            renames Available_Space_Operations.Save_Node;

   procedure Save_List (Saved_Head : in Pointers;
                        Saved_Tail : in Pointers;
                        Node_Count : in POSITIVE)
            renames Available_Space_Operations.Save_List;

pragma PAGE;
   procedure Initialize_Buffer (Buffer : in out Buffers) is

--    ---------------------
--    --declaration section
--    ---------------------
```

```
      Current_Length : INTEGER  renames Buffer.Current_Length;
      Head           : Pointers renames Buffer.Head;
      Tail           : Pointers renames Buffer.Tail;

-- ---------------------------------
-- --begin procedure Initialize_Buffer
-- ---------------------------------

   begin

      if Current_Length = -1 then

--       --handle an uninitialized buffer
         Head := New_Node;
         Tail := Head;
         Current_Length := 0;

      elsif Current_Length > 0 then

--       --handle a buffer that has something in it
         Clear_Buffer(Buffer => Buffer);

      else

--       --current length = 0 so it is already initialized
         null;

      end if;

   end Initialize_Buffer ;

pragma PAGE;
   procedure Clear_Buffer (Buffer : in out Buffers) is

--    ---------------------
--    --declaration section
--    ---------------------

      Current_Length : INTEGER  renames Buffer.Current_Length;
      Head           : Pointers renames Buffer.Head;
      Tail           : Pointers renames Buffer.Tail;

      This_Node      : Pointers;

-- ------------------------------
-- --begin procedure Clear_Buffer
-- ------------------------------

   begin

--    --make sure this is an initialized buffer
      if Current_Length = -1 then
         raise Buffer_Not_Initialized;
      end if;

--    --placed nodes in the available space list
      Save_List (Saved_Head => Head.Next,
```

```
                Saved_Tail => Tail,
                Node_Count => Current_Length);

--      --reinitialize buffer variables
        Current_Length := 0;
        Head.Next      := null;
        Tail           := Head;

    end Clear_Buffer ;

pragma PAGE;
    procedure Free_Memory is

--      ---------------------
--      --declaration section
--      ---------------------

        Node_To_Be_Freed : Pointers;

-- -----------------------------
-- --begin procedure Free_Memory
-- -----------------------------

    begin

        Clear_Out_Available_Space_List:
            while Available_Head /= Available_Tail loop

                Node_To_Be_Freed := Available_Head;
                Available_Head   := Available_Head.Next;

                Free_Node (Which_Node => Node_To_Be_Freed);

            end loop Clear_Out_Available_Space_List;

        Available_Length := 0;

    end Free_Memory ;

pragma PAGE;
    procedure Add_Element (New_Element : in     Elements;
                           Buffer      : in out Buffers) is

--      ---------------------
--      --declaration section
--      ---------------------

        Current_Length : INTEGER  renames Buffer.Current_Length;
        Tail           : Pointers renames Buffer.Tail;

        New_Tail       : Pointers;

-- -----------------------------
-- --begin procedure Add_Element
-- -----------------------------

    begin
```

```
--       --make sure buffer has been initialized
         if Current_Length = -1 then
            raise Buffer_Not_Initialized;
         end if;

--       --now get a node
         New_Tail := New_Node;

--       --now adjust the buffer
         Tail.Next      := New_Tail;
         Tail           := New_Tail;
         Tail.Data      := New_Element;
         Current_Length := Current_Length + 1;

      end Add_Element ;

pragma PAGE;
      procedure Retrieve_Element (Buffer      : in out Buffers;
                                  Old_Element :    out Elements) is

--    ---------------------
--    --declaration section
--    ---------------------

      Current_Length : INTEGER  renames Buffer.Current_Length;
      Head           : Pointers renames Buffer.Head;

      This_Node      : Pointers;

-- ---------------------------------
-- --begin procedure Retrieve_Element
-- ---------------------------------

      begin

--       --make sure an element is available
         if Current_Length = -1 then
            raise Buffer_Not_Initialized;
         elsif Current_Length = 0 then
            raise Buffer_Empty;
         end if;

--       --save dummy node in the available space list
         This_Node := Head;
         Head      := Head.Next;
         Save_Node (Saved_Node => This_Node);

--       --retrieve element (its node becomes the new dummy node)
         Old_Element := Head.Data;

--       --update buffer status
         Current_Length := Current_Length - 1;

      end Retrieve_Element ;

pragma PAGE;
```

```
   function Peek (Buffer : in  Buffers) return Elements is

--    ---------------------
--    --declaration section
--    ---------------------

      Current_Length : INTEGER  renames Buffer.Current_Length;
      Head           : Pointers renames Buffer.Head;

-- ---------------------
-- --begin function Peek
-- ---------------------

   begin

--    --make sure something is there to look at
      if Current_Length = -1 then
         raise Buffer_Not_Initialized;
      elsif Current_Length = 0 then
         raise Buffer_Empty;
      end if;

      return Head.Next.Data;

   end Peek ;

pragma PAGE;
   function Buffer_Status (Buffer : in Buffers) return Buffer_Statuses is

--    ---------------------
--    --declaration section
--    ---------------------

      Current_Length : INTEGER  renames Buffer.Current_Length;
      Status         : Buffer_Statuses;

-- -----------------------------
-- --begin function Buffer_Status
-- -----------------------------

   begin

      if Current_Length = -1 then
         Status := Uninitialized;

      elsif Current_Length = 0 then
         Status := Empty;

      else
         Status := Available;

      end if;

      return Status;

   end Buffer_Status ;
```

```
pragma PAGE;
   function Buffer_Length (Buffer : in Buffers) return NATURAL is

--     ---------------------
--     --declaration section
--     ---------------------

       Current_Length : INTEGER  renames Buffer.Current_Length;

-- ------------------------------
-- --begin function Buffer_Length
-- ------------------------------

   begin

--     --make sure the buffer has a length
       if Current_Length = -1 then
          raise Buffer_Not_Initialized;
       end if;

       return Current_Length;

   end Buffer_Length ;

pragma PAGE;
   function Dot_Next (Ptr : in Pointers) return Pointers is
   begin
      return Ptr.Next;
   end Dot_Next;

pragma PAGE;
   procedure Set_Next (Ptr          : in Pointers;
                       Ptr_Dot_Next : in Pointers) is
   begin
      Ptr.Next := Ptr_Dot_Next;
   end Set_Next;

pragma PAGE;
-------------------------------------------
--begin package Unbounded_FIFO_Buffer
--(see header for package body for details)
-------------------------------------------
begin

-- --set up available space list if one is desired
   if Initial_Available_Space_Size > 0 then

      Add_Nodes_To_Available_Space_List:
         for I in 1..Initial_Available_Space_Size loop
            Available_Tail.Next := new Nodes;
            Available_Tail      := Available_Tail.Next;
         end loop Add_Nodes_To_Available_Space_List;

      Available_Length := Initial_Available_Space_Size;

   end if;
```

```
end Unbounded_Fifo_Buffer;
```

```
separate (Abstract_Data_Structures)
package body Nonblocking_Circular_Buffer is

pragma PAGE;
   procedure Clear_Buffer (Buffer : out Buffers) is

--     ---------------------
--     --declaration section
--     ---------------------

       Head           : Buffer_Range renames Buffer.Head;
       Tail           : Buffer_Range renames Buffer.Tail;
       Current_Length : Buffer_Range renames Buffer.Current_Length;

-- ------------------------------
-- --begin procedure Clear_Buffer
-- ------------------------------

   begin

       Head           := 0;
       Tail           := 1;
       Current_Length := 0;

   end Clear_Buffer ;

pragma PAGE;
   procedure Add_Element (New_Element : in     Elements;
                          Buffer      : in out Buffers) is

--     ---------------------
--     --declaration section
--     ---------------------

       Head           : Buffer_Range renames Buffer.Head;
       Tail           : Buffer_Range renames Buffer.Tail;
       Current_Length : Buffer_Range renames Buffer.Current_Length;
       LIST           : Lists renames Buffer.LIST;

-- ------------------------------
-- --begin procedure Add_Element
-- ------------------------------

   begin

       LIST(Tail) := New_Element;

       if Head = Tail then

--         --buffer was already full and an element was overwritten; therefore,
--         --both head and tail need to be advanced, but Current_Length does
--         --not need to be changed

           if Tail = Buffer_Size then
              Head := 0;
              Tail := 0;
           else
```

```
            Head := Head + 1;
            Tail := Tail + 1;
         end if;

      else

--       --buffer was not already full; therefore, the Current_Length needs
--       --to be increment and only the tail needs to be advanced

         if Tail = Buffer_Size then
            Tail := 0;
         else
            Tail := Tail + 1;
         end if;

         Current_Length := Current_Length + 1;

      end if;

   end Add_Element ;

pragma PAGE;
   procedure Retrieve_Element (Buffer      : in out Buffers;
                               Old_Element :    out Elements) is

--    ---------------------
--    --declaration section
--    ---------------------

      Head           : Buffer_Range renames Buffer.Head;
      Tail           : Buffer_Range renames Buffer.Tail;
      Current_Length : Buffer_Range renames Buffer.Current_Length;
      LIST           : Lists        renames Buffer.LIST;

-- ----------------------------------
-- --begin procedure Retrieve_Element
-- ----------------------------------

   begin

--    --make sure there is something there to retrieve
      if Current_Length = 0 then
         raise Buffer_Empty;
      end if;

--    --advance the head to get to the next element to go out
      if Head = Buffer_Size then
         Head := 0;
      else
         Head := Head + 1;
      end if;

--    --now retrieve the element and update the state of the buffer
      Old_Element := LIST(Head);
      Current_Length := Current_Length - 1;

   end Retrieve_Element ;
```

```
pragma PAGE;
   function Peek (Buffer : in Buffers) return Elements is

--     ---------------------
--     --declaration section
--     ---------------------

       Head           : Buffer_Range renames Buffer.Head;
       Current_Length : Buffer_Range renames Buffer.Current_Length;
       LIST           : Lists        renames Buffer.LIST;

       Spot           : Buffer_Range;

-- ---------------------
-- --begin function Peek
-- ---------------------

   begin

--     --make sure there is something to peek at
       if Current_Length = 0 then
          raise Buffer_Empty;
       end if;

--     --determine location of desired element
       if Head = Buffer_Size then
          Spot := 0;
       else
          Spot := Head + 1;
       end if;

--     --return requested element
       return LIST(Spot);

   end Peek ;

pragma PAGE;
   function Buffer_Status (Buffer : in Buffers) return Buffer_Statuses is

--     ---------------------
--     --declaration section
--     ---------------------

       Current_Length : Buffer_Range renames Buffer.Current_Length;

       Status         : Buffer_Statuses;

-- ----------------------------
-- --begin function Buffer_Status
-- ----------------------------

   begin

       if Current_Length = 0 then
          Status := Empty;
       elsif Current_Length = Buffer_Size then
```

```
            Status := Full;
         else
            Status := Available;
         end if;

         return Status;

      end Buffer_Status ;

   pragma PAGE;
      function Buffer_Length (Buffer : in Buffers) return Buffer_Range is

      begin

         return Buffer.Current_Length;

      end Buffer_Length ;

   end Nonblocking_Circular_Buffer;
```

```
with Unchecked_Deallocation;
separate (Abstract_Data_Structures)
package body Unbounded_Priority_Queue is

-- ---------------------
-- --declaration section
-- ---------------------

-- --this pointers is accessed ONLY when setting up the Available_Space
   Initial_Head    : Pointers := new Nodes;

   Available_Space : Queues := (Current_Length => 0,
                                Head           => Initial_Head,
                                Tail           => Initial_Head);

   Available_Length : INTEGER  renames Available_Space.Current_Length;
   Available_Head   : Pointers renames Available_Space.Head;
   Available_Tail   : Pointers renames Available_Space.Tail;

   procedure Free is new Unchecked_Deallocation
                         (Object => Nodes,
                          NAME   => Pointers);

   procedure Free_Node (Which_Node : in out Pointers)
             renames Free;

   function Dot_Next (Ptr : in Pointers) return Pointers;

   procedure Set_Next (Ptr          : in Pointers;
                       Ptr_Dot_Next : in Pointers);

   package Available_Space_Operations is new
           Available_Space_List_Operations
              (Nodes                => Nodes,
               Pointers             => Pointers,
               Available_Length     => Available_Length,
               Available_Head       => Available_Head,
               Available_Tail       => Available_Tail);

   function New_Node return Pointers
            renames Available_Space_Operations.New_Node;

   procedure Save_Node (Saved_Node : in Pointers)
            renames Available_Space_Operations.Save_Node;

   procedure Save_List (Saved_Head : in Pointers;
                        Saved_Tail : in Pointers;
                        Node_Count : in POSITIVE)
            renames Available_Space_Operations.Save_List;

pragma PAGE;
   procedure Initialize (Queue : in out Queues) is

--    ---------------------
--    --declaration section
--    ---------------------
```

```
      Current_Length : INTEGER  renames Queue.Current_Length;
      Head           : Pointers renames Queue.Head;
      Tail           : Pointers renames Queue.Tail;

-- ----------------------------
-- --begin procedure Initialize
-- ----------------------------

   begin

      if Current_Length = -1 then

--       --handle an uninitialized queue
         Head := New_Node;
         Tail := Head;
         Current_Length := 0;

      elsif Current_Length > 0 then

--       --handle a queue that has something in it
         Clear_Queue(Queue => Queue);

      else

--       --current length = 0 so it is already initialized
         null;

      end if;

   end Initialize ;

pragma PAGE;
   procedure Clear_Queue (Queue : in out Queues) is

--    ---------------------
--    --declaration section
--    ---------------------

      Current_Length : INTEGER  renames Queue.Current_Length;
      Head           : Pointers renames Queue.Head;
      Tail           : Pointers renames Queue.Tail;

      This_Node      : Pointers;

-- ----------------------------
-- --begin procedure Clear_Queue
-- ----------------------------

   begin

--    --make sure this is an initialized queue
      if Current_Length = -1 then

         raise Queue_Not_Initialized;

      elsif Current_Length > 0 then
```

```
--          --placed nodes in the available space list
            Save_List (Saved_Head => Head.Next,
                       Saved_Tail => Tail,
                       Node_Count => Current_Length);

--          --reinitialize queue variables
            Current_Length := 0;
            Head.Next      := null;
            Tail           := Head;

         end if;

      end Clear_Queue ;

pragma PAGE;
      procedure Free_Memory is

--       ---------------------
--       --declaration section
--       ---------------------

         Node_To_Be_Freed : Pointers;

-- ----------------------------
-- --begin procedure Free_Memory
-- ----------------------------

      begin

         Clear_Out_Available_Space_List:
            while Available_Head /= Available_Tail loop
               Node_To_Be_Freed := Available_Head;
               Available_Head   := Available_Head.Next;
               Free_Node (Which_Node => Node_To_Be_Freed);
            end loop Clear_Out_Available_Space_List;

         Available_Length := 0;

      end Free_Memory ;

pragma PAGE;
      procedure Add_Element (New_Element  : in     Elements;
                             New_Priority : in     Priorities;
                             Queue        : in out Queues) is

--       ---------------------
--       --declaration section
--       ---------------------

         Current_Length : INTEGER  renames Queue.Current_Length;
         Head           : Pointers renames Queue.Head;
         Tail           : Pointers renames Queue.Tail;

         Before         : Pointers;
         Here           : Pointers;

-- ----------------------------
```

```
-- --begin procedure Add_Element
-- ------------------------------

   begin

--     --make sure queue has been initialized
       if Current_Length = -1 then
          raise Queue_Not_Initialized;
       end if;

--     --find the nodes which are to go before and after the new element
       Before := Head;
       loop
          exit when (Before = Tail) or else
                    (New_Priority > Before.Next.PRIORITY);
          Before := Before.Next;
       end loop;

--     --now get a new node
       Here := New_Node;

--     --set up the new node
       Here.PRIORITY := New_Priority;
       Here.Data     := New_Element;
       Here.Next     := Before.Next;
       Before.Next   := Here;

--     --readjust the tail, if required
       if Before = Tail then
          Tail := Here;
       end if;

--     --now adjust the queue
       Current_Length := Current_Length + 1;

   end Add_Element ;

pragma PAGE;
   procedure Retrieve_Element (Queue       : in out Queues;
                               Old_Element :    out Elements) is

--     --------------------
--     --declaration section
--     ---------------------

       Current_Length : INTEGER  renames Queue.Current_Length;
       Head           : Pointers renames Queue.Head;

       This_Node      : Pointers;

-- ---------------------------------
-- --begin procedure Retrieve_Element
-- ---------------------------------

   begin

--     --make sure an element is available
```

```
      if Current_Length = -1 then
         raise Queue_Not_Initialized;
      elsif Current_Length = 0 then
         raise Queue_Empty;
      end if;

--    --save dummy node in the available space list
      This_Node := Head;
      Head      := Head.Next;
      Save_Node (Saved_Node => This_Node);

--    --retrieve element (its node becomes the new dummy node)
      Old_Element := Head.Data;

--    --update queue status
      Current_Length := Current_Length - 1;

   end Retrieve_Element ;

pragma PAGE;
   function Peek (Queue : in Queues) return Elements is

--    ---------------------
--    --declaration section
--    ---------------------

      Current_Length : INTEGER  renames Queue.Current_Length;
      Head           : Pointers renames Queue.Head;

-- ---------------------
-- --begin function Peek
-- ---------------------

   begin

--    --make sure something is there to look at
      if Current_Length = -1 then
         raise Queue_Not_Initialized;
      elsif Current_Length = 0 then
         raise Queue_Empty;
      end if;

      return Head.Next.Data;

   end Peek ;

pragma PAGE;
   function Queue_Status (Queue : in Queues) return Queue_Statuses is

--    ---------------------
--    --declaration section
--    ---------------------

      Current_Length : INTEGER  renames Queue.Current_Length;
      Status         : Queue_Statuses;

-- ---------------------------
```

```
-- --begin function Queue_Status
-- ----------------------------

   begin

      if Current_Length = -1 then
         Status := Uninitialized;
      elsif Current_Length = 0 then
         Status := Empty;
      else
         Status := Available;
      end if;

      return Status;

   end Queue_Status ;

pragma PAGE;
   function Queue_Length (Queue : in Queues) return NATURAL is

--    ---------------------
--    --declaration section
--    ---------------------

      Current_Length : INTEGER  renames Queue.Current_Length;

-- ----------------------------
-- --begin function Queue_Length
-- ----------------------------

   begin

--    --make sure the queue has a length
      if Current_Length = -1 then
         raise Queue_Not_Initialized;
      end if;

      return Current_Length;

   end Queue_Length ;

pragma PAGE;
   function Dot_Next (Ptr : in Pointers) return Pointers is
   begin
      return Ptr.Next;
   end Dot_Next;

pragma PAGE;
   procedure Set_Next (Ptr          : in Pointers;
                       Ptr_Dot_Next : in Pointers) is
   begin
      Ptr.Next := Ptr_Dot_Next;
   end Set_Next;

pragma PAGE;
-------------------------------------------
--begin package Unbounded_Priority_Queue
```

```
-- (see header for package body for details)
-------------------------------------------------
begin

-- --set up available space list if one is desired
   if Initial_Available_Space_Size > 0 then

      Add_Nodes_To_Available_Space_List:
         for I in 1..Initial_Available_Space_Size loop
            Available_Tail.Next := new Nodes;
            Available_Tail      := Available_Tail.Next;
         end loop Add_Nodes_To_Available_Space_List;

      Available_Length    := Initial_Available_Space_Size;

   end if;

end Unbounded_Priority_Queue;
```

```
separate (Abstract_Data_Structures)
package body Bounded_Stack is

pragma PAGE;
   procedure Clear_Stack (Stack : out Stacks) is

   begin

      Stack.Top := 0;

   end Clear_Stack ;

pragma PAGE;
   procedure Add_Element (New_Element : in     Elements;
                          Stack       : in out Stacks) is

--    ---------------------
--    --declaration section
--    ---------------------

      LIST : Lists               renames Stack.LIST;
      Top  : Stack_Length_Range renames Stack.Top;

-- -----------------------------
-- --begin procedure Add_Element
-- -----------------------------

   begin

--    --make sure the stack is not already full
      if Top = Stack_Size then
         raise Stack_Full;
      end if;

--    --add element to the stack
      Top       := Top + 1;
      LIST(Top) := New_Element;

   end Add_Element ;

pragma PAGE;
   procedure Retrieve_Element (Stack       : in out Stacks;
                               Old_Element :    out Elements) is

--    ---------------------
--    --declaration section
--    ---------------------

      LIST : Lists               renames Stack.LIST;
      Top  : Stack_Length_Range renames Stack.Top;

-- ---------------------------------
-- --begin procedure Retrieve_Element
-- ---------------------------------

   begin
```

```
--      --make sure there is something in the stack to retrieve
        if Top = 0 then
           raise Stack_Empty;
        end if;

--      --retrieve and remove the top element from the stack
        Old_Element := LIST(Top);
        Top         := Top - 1;

    end Retrieve_Element ;

pragma PAGE;
    function Peek (Stack : in Stacks) return Elements is

--      ---------------------
--      --declaration section
--      ---------------------

        LIST : Lists                renames Stack.LIST;
        Top  : Stack_Length_Range renames Stack.Top;

-- ---------------------
-- --begin function Peek
-- ---------------------

    begin

--      --make sure there is something in the stack
        if Top = 0 then
           raise Stack_Empty;
        end if;

--      --return value in top element of the stack
        return LIST(Top);

    end Peek ;

pragma PAGE;
    function Stack_Status (Stack : in Stacks) return Stack_Statuses is

--      ---------------------
--      --declaration section
--      ---------------------

        Top    : Stack_Length_Range renames Stack.Top;

        Status : Stack_Statuses;

-- ---------------------------
-- --begin function Stack_Status
-- ---------------------------

    begin

        if Top = 0 then
           Status := Empty;
        elsif Top = Stack_Size then
```

```
            Status := Full;
         else
            Status := Available;
         end if;

         return Status;

      end Stack_Status ;

   pragma PAGE;
      function Stack_Length (Stack : in Stacks) return Stack_Length_Range is

      begin

         return Stack.Top;

      end Stack_Length ;

   end Bounded_Stack;
```

```
with Unchecked_Deallocation;
separate (Abstract_Data_Structures)
package body Unbounded_Stack is

-- ---------------------
-- --declaration section
-- ---------------------

-- --this pointer is accessed ONLY when setting up the Available_Space
   Initial_Head : Pointers := new Nodes;

   Available_Space : Stacks := (Current_Length => 0,
                                Top            => Initial_Head,
                                Bottom         => Initial_Head);

   Available_Length : INTEGER  renames Available_Space.Current_Length;
   Available_Top    : Pointers renames Available_Space.Top;
   Available_Bottom : Pointers renames Available_Space.Bottom;

   procedure Free is new Unchecked_Deallocation
                         (Object => Nodes,
                          NAME   => Pointers);

   procedure Free_Node (Which_Node : in out Pointers)
             renames Free;

   function Dot_Next (Ptr : in Pointers) return Pointers;

   procedure Set_Next (Ptr          : in Pointers;
                       Ptr_Dot_Next : in Pointers);

   package Available_Space_Operations is new
           Available_Space_List_Operations
              (Nodes                => Nodes,
               Pointers             => Pointers,
               Available_Length     => Available_Length,
               Available_Head       => Available_Top,
               Available_Tail       => Available_Bottom);

   function New_Node return Pointers
            renames Available_Space_Operations.New_Node;

   procedure Save_Node (Saved_Node : in Pointers)
            renames Available_Space_Operations.Save_Node;

   procedure Save_List (Saved_Head : in Pointers;
                        Saved_Tail : in Pointers;
                        Node_Count : in POSITIVE)
            renames Available_Space_Operations.Save_List;

pragma PAGE;
   procedure Initialize (Stack : in out Stacks) is

--    ---------------------
--    --declaration section
--    ---------------------
```

```
      Current_Length : INTEGER  renames Stack.Current_Length;
      Top            : Pointers renames Stack.Top;
      Bottom         : Pointers renames Stack.Bottom;

-- ----------------------------
-- --begin procedure Initialize
-- ----------------------------

   begin

      if Current_Length = -1 then

--       --handle an uninitialized stack
         Top            := New_Node;
         Bottom         := Top;
         Current_Length := 0;

      elsif Current_Length > 0 then

--       --handle a stack that has elements in it
         Clear_Stack (Stack => Stack);

      else

--       --current length = 0, so do nothing
         null;

      end if;

   end Initialize ;

pragma PAGE;
   procedure Clear_Stack (Stack : in out Stacks) is

--    ---------------------
--    --declaration section
--    ---------------------

      Current_Length : INTEGER  renames Stack.Current_Length;
      Top            : Pointers renames Stack.Top;
      Bottom         : Pointers renames Stack.Bottom;

      This_Node      : Pointers;

-- ----------------------------
-- --begin procedure Clear_Stack
-- ----------------------------

   begin

--    --make sure stack has been initialized
      if Current_Length = -1 then

         raise Stack_Not_Initialized;

--    --make sure there is something in the stack
      elsif Current_Length /= 0 then
```

```
--          --placed nodes in the available space list
            Save_List (Saved_Head => Top.Next,
                       Saved_Tail => Bottom,
                       Node_Count => Current_Length);

--          --reinitialize stack variables
            Top.Next       := null;
            Bottom         := Top;
            Current_Length := 0;

         end if;

      end Clear_Stack ;

pragma PAGE;
      procedure Free_Memory is

--       ---------------------
--       --declaration section
--       ---------------------

         This_Node : Pointers;

-- ----------------------------
-- --begin procedure Free_Memory
-- ----------------------------

      begin

         Deallocate_Nodes_In_Available_Space_List:
            while Available_Top /= Available_Bottom loop

               This_Node := Available_Top;
               Available_Top := Available_Top.Next;
               Free_Node (Which_Node => This_Node);

            end loop Deallocate_Nodes_In_Available_Space_List;

         Available_Length   := 0;
         Available_Top.Next := null;

      end Free_Memory ;

pragma PAGE;
      procedure Add_Element (New_Element : in     Elements;
                             Stack       : in out Stacks) is

--       ---------------------
--       --declaration section
--       ----------------- --

         Current_Length : INTEGER  renames Stack.Current_Length;
         Top            : Pointers renames Stack.Top;

         Ptr            : Pointers;
```

```
-- ------------------------------
-- --begin procedure Add_Element
-- ------------------------------

   begin

      if Current_Length = -1 then
         raise Stack_Not_Initialized;
      end if;

--    --get a node and initialize it
      Ptr       := New_Node;
      Ptr.Data := New_Element;

--    --place the node on the stack
      Ptr.Next        := Top;
      Top             := Ptr;
      Current_Length := Current_Length + 1;

   end Add_Element ;

pragma PAGE;
   procedure Retrieve_Element (Stack       : in out Stacks;
                               Old_Element :    out Elements) is

--    ---------------------
--    --declaration section
--    ---------------------

      Current_Length : INTEGER  renames Stack.Current_Length;
      Top            : Pointers renames Stack.Top;

      This_Node      : Pointers;

-- ---------------------------------
-- --begin procedure Retrieve_Element
-- ---------------------------------

   begin

--    --make sure there is something to retrieve
      if Current_Length = -1 then
         raise Stack_Not_Initialized;
      elsif Current_Length = 0 then
         raise Stack_Empty;
      end if;

--    --retrieve data in the top node
      Old_Element := Top.Data;

--    --dispose of top node and adjust the stack
      This_Node := Top;
      Top       := Top.Next;
      Save_Node (Saved_Node => This_Node);
      Current_Length := Current_Length - 1;

   end Retrieve_Element ;
```

```
pragma PAGE;
   function Peek (Stack : in Stacks) return Elements is

--     ---------------------
--     --declaration section
--     ---------------------

       Current_Length : INTEGER  renames Stack.Current_Length;
       Top            : Pointers renames Stack.Top;

-- ---------------------
-- --begin function Peek
-- ---------------------

   begin

--     --make sure there is something to peek at
       if Current_Length = -1 then
          raise Stack_Not_Initialized;
       elsif Current_Length = 0 then
          raise Stack_Empty;
       end if;

--     --returned desired element
       return Top.Data;

   end Peek ;

pragma PAGE;
   function Stack_Status (Stack : in Stacks) return Stack_Statuses is

--     ---------------------
--     --declaration section
--     ---------------------

       Current_Length : INTEGER  renames Stack.Current_Length;

       Status         : Stack_Statuses;

-- ----------------------------
-- --begin function Stack_Status
-- ----------------------------

   begin

      if Current_Length = -1 then
         Status := Uninitialized;

      elsif Current_Length = 0 then
         Status := Empty;

      else
         Status := Available;

      end if;
```

```
      return Status;

   end Stack_Status ;

pragma PAGE;
   function Stack_Length (Stack : in Stacks) return NATURAL is

--     ---------------------
--     --declaration section
--     ---------------------

      Current_Length : INTEGER  renames Stack.Current_Length;

-- ----------------------------
-- --begin function Stack_Length
-- ----------------------------

   begin

--     --make sure stack has been initialized
      if Current_Length = -1 then
         raise Stack_Not_Initialized;
      end if;

      return Current_Length;

   end Stack_Length ;

pragma PAGE;
   function Dot_Next (Ptr : in Pointers) return Pointers is
   begin
      return Ptr.Next;
   end Dot_Next;

pragma PAGE;
   procedure Set_Next (Ptr          : in Pointers;
                       Ptr_Dot_Next : in Pointers) is
   begin
      Ptr.Next := Ptr_Dot_Next;
   end Set_Next;

pragma PAGE;
-----------------------------------------
--begin package Unbounded_Stack
--(see header for package body for details)
-----------------------------------------
begin

-- --set up available space list if one is desired
   if Initial_Available_Space_Size > 0 then

      Add_Nodes_To_Available_Space_List:
         for I in 1..Initial_Available_Space_Size loop
            Available_Bottom.Next := new Nodes;
            Available_Bottom      := Available_Bottom.Next;
         end loop Add_Nodes_To_Available_Space_List;
```

```
      Available_Length := Initial_Available_Space_Size;
   end if;
end Unbounded_Stack;
```

3.3.8 GENERAL UTILITIES

(This page intentionally left blank.)

THIS REPORT HAS BEEN DELIMITED
AND CLEARED FOR PUBLIC RELEASE
UNDER DOD DIRECTIVE 5200.20 AND
NO RESTRICTIONS ARE IMPOSED UPON
ITS USE AND DISCLOSURE.

DISTRIBUTION STATEMENT A

APPROVED FOR PUBLIC RELEASE;
DISTRIBUTION UNLIMITED.

3.3.8.1 GENERAL_UTILITIES TLCSC P361 (CATALOG #P267-0)

This package provides a group of general utility routines used in a missile system.

The decomposition for this part is the same as that shown in the Top-Level Design Document.

3.3.8.1.1 REQUIREMENTS ALLOCATION

This part meets requirement R141.

3.3.8.1.2 LOCAL ENTITIES DESIGN

None.

3.3.8.1.3 INPUT/OUTPUT

None.

3.3.8.1.4 LOCAL DATA

None.

3.3.8.1.5 PROCESS CONTROL

Not applicable.

3.3.8.1.6 PROCESSING

The following describes the processing performed by this part:

```
package body General_Utilities is

end General_Utilities;
```

3.3.8.1.7 UTILIZATION OF OTHER ELEMENTS

None.

3.3.8.1.8 LIMITATIONS

None.

3.3.8.1.9 LLCSC DESIGN

None.

3.3.8.1.10 UNIT DESIGN

3.3.8.1.10.1 INSTRUCTION_SET_TEST UNIT DESIGN (CATALOG #P268-0)

This part is a generic function which checks for proper processor operation by executing a function and comparing the result to the expected result. If the expected and derived values match, "True" is returned. The part's generic parameter may be any type, but a Test function must be supplied which matches the parameter defined in the specification.

3.3.8.1.10.1.1 REQUIREMENTS ALLOCATION

This part meets requirement R141.

3.3.8.1.10.1.2 LOCAL ENTITIES DESIGN

None.

3.3.8.1.10.1.3 INPUT/OUTPUT

GENERIC PARAMETERS:

Data types:

The following table summarizes the generic formal types required by this part:

Name	Type	Description
Return_Values	private	May be any type. The type which the included function must return.

Subprograms:

The following table summarizes the generic formal subroutines required by this part:

Name	Type	Description
Test	function	the function to be tested, it must return a value of Return_Values type.

FORMAL PARAMETERS:

The following table describes this part's formal parameters:

Name	Type	Mode	Description
Correct_Answer	Return_Values	in	The answer which is to be compared to what the function returns.

3.3.8.1.10.1.4 LOCAL DATA

None.

3.3.8.1.10.1.5 PROCESS CONTROL

Not applicable.

3.3.8.1.10.1.6 PROCESSING

The following describes the processing performed by this part:

```
   function Instruction_Set_Test( Correct_Answer : Return_Values )
                                        return BOOLEAN is

   begin
      return Test = Correct_Answer;
--    -- returns true if function and answer are the same
--    -- false if they are not
   end Instruction_Set_Test;
```

3.3.8.1.10.1.7 UTILIZATION OF OTHER ELEMENTS

None.

3.3.8.1.10.1.8 LIMITATIONS

None.

(This page left intentionally blank.)

```
package body General_Utilities is

   function Instruction_Set_Test( Correct_Answer : Return_Values )
                                        return BOOLEAN is

   begin
      return Test = Correct_Answer;
--    -- returns true if function and answer are the same
--    -- false if they are not
   end Instruction_Set_Test;

end General_Utilities;
```

(This page left intentionally blank.)

3.3.8.2 COMMUNICATION_PARTS TLCSC P602 (CATALOG #P691-0)

This package provides a group of communication routines used in a missile system.

The decomposition for this part is the same as that shown in the Top-Level Design Document.

3.3.8.2.1 REQUIREMENTS ALLOCATION

This part meets requirement R137.

3.3.8.2.2 LOCAL ENTITIES DESIGN

None.

3.3.8.2.3 INPUT/OUTPUT

None.

3.3.8.2.4 LOCAL DATA

None.

3.3.8.2.5 PROCESS CONTROL

Not applicable.

3.3.8.2.6 PROCESSING

The following describes the processing performed by this part:

```
package body Communication_Parts is

end Communication_Parts;
```

3.3.8.2.7 UTILIZATION OF OTHER ELEMENTS

None.

3.3.8.2.8 LIMITATIONS

None.

3.3.8.2.9 LLCSC DESIGN

3.3.8.2.9.1 UPDATE_EXCLUSION PACKAGE DESIGN (CATALOG #P692-0)

This part is a generic package containing a task providing a mechanism for ensuring that data accessed by more than one asymchronous task is properly protected for such accesses. The part's generic parameter can be any type.

The decomposition for this part is the same as that shown in the Top-Level Design Document.

3.3.8.2.9.1.1 REQUIREMENTS ALLOCATION

This part meets requirement R137.

3.3.8.2.9.1.2 LOCAL ENTITIES DESIGN

None.

3.3.8.2.9.1.3 INPUT/OUTPUT

GENERIC PARAMETERS:

Data types:

The following table summarizes the generic formal types required by this part:

Name	Type	Description
Element_Type	private	Allows any type to be protected

Data objects:

The following table summarizes the generic formal objects required by this part:

Name	Type	Description
Initial_Value	Element_Type	Allows the data type to be initialized so that the first time Start_Update_Request is called a constraint error is not raised by some uninitialized value.

FORMAL PARAMETERS:

The following table describes the formal parameters for the task entries in the task contained in this part.

Task	Name	Mode	Type	Description
Read_ Update	Read_Request	Output	Element_Type	Contains the value of the returned data.
	Start_Update_ Request	Output	Element_Type	Contains the value of the returned data.
	Complete_ Update_ Request	Input	Element_Type	Contains the new value of the data to replace the protected data.

3.3.8.2.9.1.4 LOCAL DATA

None.

3.3.8.2.9.1.5 PROCESS CONTROL

Not applicable.

3.3.8.2.9.1.6 PROCESSING

The following describes the processing performed by this part:

```
package body Update_Exclusion is

   task Read_Update is
      entry Task_Read_Request( Requested_Data : out Element_Type );
      entry Task_Start_Update_Request( Old_Data : out Element_Type );
      entry Task_Complete_Update_Request( New_Data : in Element_Type );
   end Read_Update;

   procedure Attempt_Read( Requested_Data : in out Element_Type;
                           Result         : out Rendezvous_Flags ) is
   begin
      select
         Read_Update.Task_Read_Request( Requested_Data );
         Result := Success;
      Else
         Result := Failure;
      end select;
   end Attempt_Read;

   procedure Attempt_Read_Wait( Requested_Data : in out Element_Type;
                                Result         : out Rendezvous_Flags ) is
   begin
      Read_Update.Task_Read_Request( Requested_Data );
      Result := Success;
   end Attempt_Read_Wait;

   procedure Attempt_Read_Delay( Requested_Data : in out Element_Type;
                                 Result         : out Rendezvous_Flags;
                                 Delay_Time     : in  DURATION )       is
```

```
begin
   Result := Failure;
   select
      Read_Update.Task_Read_Request( Requested_Data );
      Result := Success;
   or
      DELAY Delay_Time;
   end select;
end Attempt_Read_Delay;

procedure Attempt_Start_Update( Old_Data : in out Element_Type;
                                New_Id   : out Rendezvous_Ids;
                                Result   : out Rendezvous_Flags) is
begin
   select
      Read_Update.Task_Start_Update_Request( Old_Data => Old_Data );
      New_Id := Id;
      Result := Success;
   else
      Result := Failure;
      New_Id := 0;
   end select;
end Attempt_Start_Update;

procedure Attempt_Start_Update_Wait( Old_Data : in out Element_Type;
                                     New_Id   : out Rendezvous_Ids;
                                     Result   : out Rendezvous_Flags ) is
begin
   Read_Update.Task_Start_Update_Request( Old_Data => Old_Data );
   New_Id := Id;
   Result := Success;
end Attempt_Start_Update_Wait;

procedure Attempt_Start_Update_Delay( Old_Data : in out Element_Type;
                                      New_Id   : out Rendezvous_Ids;
                                      Result   : out Rendezvous_Flags;
                                      Time     : in  DURATION )  is
begin
   Result := Failure;
   select
      Read_Update.Task_Start_Update_Request( Old_Data => Old_Data );
      New_Id := Id;
      Result := Success;
   or
      DELAY Time;
   end select;
end Attempt_Start_Update_Delay;

procedure Attempt_Complete_Update( New_Data  : in  Element_Type;
                                   Passed_Id : in  Rendezvous_Ids;
                                   Result    : out Rendezvous_Flags ) is
begin
   if Passed_Id = Id then
      select
         Read_Update.Task_Complete_Update_Request( New_Data );
         Result := Success;
      else
```

```
            Result := Failure;
         end select;
      else
         Result := Bad_Id;
      end if;
   end Attempt_Complete_Update;

   task body Read_Update is

      Protected_Data : Element_Type := Initial_Value;

   begin
      process_continually:
         loop
            select
               accept Task_Read_Request (Requested_Data : out Element_Type)
               do
                  Requested_Data := Protected_Data;
               end Task_Read_Request;
            or
               accept Task_Start_Update_Request (Old_Data : out Element_Type)
               do
                  Old_Data := Protected_Data;
               end Task_Start_Update_Request;

               accept Task_Complete_Update_Request (New_Data : in Element_Type)
               do
                  Protected_Data := New_Data;
               end Task_Complete_Update_Request;

               if Id = Rendezvous_Ids'LAST then
                  Id := Rendezvous_Ids'FIRST + 1;
               else
                  Id := Rendezvous_Ids'SUCC( Id );
               end if;
            or
               terminate;
            end select;
         end loop process_continually;
   end Read_Update;

end Update_Exclusion;
```

3.3.8.2.9.1.7 UTILIZATION OF OTHER ELEMENTS

None.

3.3.8.2.9.1.8 LIMITATIONS

None.

3.3.8.2.9.1.9 LLCSC DESIGN

None.

3.3.8.2.9.1.10 UNIT DESIGN

None.

3.3.8.2.10 UNIT DESIGN

None.

```
package body Communication_Parts is

   package body Update_Exclusion is

      task Read_Update is
         entry Task_Read_Request( Requested_Data : out Element_Type );
         entry Task_Start_Update_Request( Old_Data : out Element_Type );
         entry Task_Complete_Update_Request( New_Data : in Element_Type );
      end Read_Update;

      procedure Attempt_Read( Requested_Data : in out Element_Type;
                              Result         : out Rendezvous_Flags ) is
      begin
         select
            Read_Update.Task_Read_Request( Requested_Data );
            Result := Success;
         else
            Result := Failure;
         end select;
      end Attempt_Read;

      procedure Attempt_Read_Wait( Requested_Data : in out Element_Type;
                                   Result         : out Rendezvous_Flags ) is
      begin
         Read_Update.Task_Read_Request( Requested_Data );
         Result := Success;
      end Attempt_Read_Wait;

      procedure Attempt_Read_Delay( Requested_Data : in out Element_Type;
                                    Result         : out Rendezvous_Flags;
                                    Delay_Time     : in  DURATION )       is
      begin
         Result := Failure;
         select
            Read_Update.Task_Read_Request( Requested_Data );
            Result := Success;
         or
            delay Delay_Time;
         end select;
      end Attempt_Read_Delay;

      procedure Attempt_Start_Update( Old_Data : in out Element_Type;
                                      New_Id   : out Rendezvous_Ids;
                                      Result   : out Rendezvous_Flags) is
      begin
         select
            Read_Update.Task_Start_Update_Request( Old_Data => Old_Data );
            New_Id := Id;
            Result := Success;
         else
            Result := Failure;
            New_Id := 0;
         end select;
      end Attempt_Start_Update;

      procedure Attempt_Start_Update_Wait( Old_Data : in out Element_Type;
                                           New_Id   : out Rendezvous_Ids;
```

```
                                        Result    : out Rendezvous_Flags ) is
   begin
      Read_Update.Task_Start_Update_Request( Old_Data => Old_Data );
      New_Id := Id;
      Result := Success;
   end Attempt_Start_Update_Wait;

   procedure Attempt_Start_Update_Delay( Old_Data : in out Element_Type;
                                         New_Id   : out Rendezvous_Ids;
                                         Result   : out Rendezvous_Flags;
                                         Time     : in  DURATION ) is
   begin
      Result := Failure;
      select
         Read_Update.Task_Start_Update_Request( Old_Data => Old_Data );
         New_Id := Id;
         Result := Success;
      or
         delay Time;
      end select;
   end Attempt_Start_Update_Delay;

   procedure Attempt_Complete_Update( New_Data  : in  Element_Type;
                                      Passed_Id : in  Rendezvous_Ids;
                                      Result    : out Rendezvous_Flags ) is
   begin
      if Passed_Id = Id then
         select
            Read_Update.Task_Complete_Update_Request( New_Data );
            Result := Success;
         else
            Result := Failure;
         end select;
      else
         Result := Bad_Id;
      end if;
   end Attempt_Complete_Update;

   task body Read_Update is

      Protected_Data : Element_Type := Initial_Value;

   begin
      Process_Continually:
         loop
            select
               accept Task_Read_Request (Requested_Data : out Element_Type)
               do
                  Requested_Data := Protected_Data;
               end Task_Read_Request;
            or
               accept Task_Start_Update_Request (Old_Data : out Element_Type)
               do
                  Old_Data := Protected_Data;
               end Task_Start_Update_Request;

               accept Task_Complete_Update_Request (New_Data : in Element_Type)
```

```
                    do
                       Protected_Data := New_Data;
                    end Task_Complete_Update_Request;

                    if Id = Rendezvous_Ids'LAST then
                       Id := Rendezvous_Ids'FIRST + 1;
                    else
                       Id := Rendezvous_Ids'SUCC( Id );
                    end if;
                 or
                    terminate;
                 end select;
              end loop Process_Continually;
        end Read_Update;

     end Update_Exclusion;

end Communication_Parts;
```

(This page left intentionally blank.)

3.3.9 EQUIPMENT INTERFACES

(This page intentionally left blank.)

3.3.9.1 CLOCK_HANDLER TLCSC P634 (CATALOG #P270-0)

This package contains the routines required to maintain an internal clock.

The following routines are provided to manipulate the clock: o Reset clock (effectively zeroes out the clock) o Synchronize clock (effectively sets the clock to the specified time) o Current time (effectively reads the internal clock)

In addition, a Converted_Time routine is provided to convert a CALENDAR. TIME to the "local time zone".

An Elapsed_Time routine is provided to act as a stopwatch. It returns the elapsed time between successive calls to the function. This function is not affected by resetting or synchronizing the clock.

The decomposition for this part is the same as that shown in the Top-Level Design Document.

3.3.9.1.1 REQUIREMENTS ALLOCATION

This part meets CAMP requirement R046.

3.3.9.1.2 LOCAL ENTITIES DESIGN

None.

3.3.9.1.3 INPUT/OUTPUT

GENERIC PARAMETERS:

This part is a parameterless generic.

3.3.9.1.4 LOCAL DATA

Data objects:

The following table describes the data objects maintained by this part:

Name	Type	Value	Description
Reference_Time	CALENDAR.TIME	N/A	Internal reference clock maintained by this part
Time_Last_Called	CALENDAR.TIME	N/A	Last time the Elapsed_ Time function was called

3.3.9.1.5 PROCESS CONTROL

Not applicable.

3.3.9.1.6 PROCESSING

The following describes the processing performed by this part:

```
package body Clock_Handler is

   use CALENDAR;

-- --------------------
-- --local declarations
-- --------------------

   Reference_Time       : CALENDAR.TIME := CALENDAR.CLOCK;
   Time_Last_Called     : CALENDAR.TIME := CALENDAR.CLOCK;

end Clock_Handler;
```

3.3.9.1.7 UTILIZATION OF OTHER ELEMENTS

The following library units were with'd by the package specification of this part:

1. CALENDAR

UTILIZATION OF EXTERNAL ELEMENTS:

Subprograms and task entries:

The following table summarizes the external subroutines and task entries required by this part:

Name	Type	Source	Description
Clock	Function	Calendar	Returns the internal system time

Data types:

The following table summarizes the external types required by this part:

Name	Type	Source	Description
TIME	private	CALENDAR	Implementation-dependent representation of time
DURATION	fixed	STANDARD	Represents a length of time

3.3.9.1.8 LIMITATIONS

The following table describes the exceptions propagated by this part:

Name	When/Why Raised
STANDARD.TIME_ERROR	Raised by the following routines if a difference in times does not fit within the range of type STANDARD.DURATION: o Current_Time o Converted_Time o Elapsed_Time o Synchronize_Time

3.3.9.1.9 LLCSC DESIGN

None.

3.3.9.1.10 UNIT DESIGN

3.3.9.1.10.1 CURRENT_TIME (FUNCTION BODY) UNIT DESIGN

This function returns the time of the current time of the clock. The current time is the time which has passed since the last time the internal clock was reset or since the time specified when the clock was synchronized.

3.3.9.1.10.1.1 REQUIREMENTS ALLOCATION

This part partially meets requirement CAMP R046.

3.3.9.1.10.1.2 LOCAL ENTITIES DESIGN

None.

3.3.9.1.10.1.3 INPUT/OUTPUT

None.

3.3.9.1.10.1.4 LOCAL DATA

None.

3.3.9.1.10.1.5 PROCESS CONTROL

Not applicable.

3.3.9.1.10.1.6 PROCESSING

The following describes the processing performed by this part:

```
function Current_Time return DURATION is

begin

   return CALENDAR.CLOCK - Reference_Time;

end Current_Time;
```

3.3.9.1.10.1.7 UTILIZATION OF OTHER ELEMENTS

The following library units were previously with'd and are visible to this part:

1. Calendar

UTILIZATION OF EXTERNAL ELEMENTS:

Subprograms and task entries:

The following table summarizes the external subroutines and task entries required by this part:

Name	Type	Source	Description
Clock	Function	Calendar	Returns the internal system time

Data types:

The following table summarizes the external types required by this part:

Name	Type	Source	Description
TIME	private	CALENDAR	Implementation-dependent representation of time
DURATION	fixed	STANDARD	Represents a length of time

UTILIZATION OF OTHER ELEMENTS IN TOP LEVEL COMPONENT:

The following tables describe the elements used by this part but defined elsewhere in the parent top level component:

Data objects:

The following table summarizes the objects required by this part and defined in the package body of Clock_Handler:

Name	Type	Value	Description
Reference_Time	CALENDAR.TIME	N/A	Internal reference clock maintained by this part

3.3.9.1.10.1.8 LIMITATIONS

The following table describes the exceptions propagated by this part:

Name	When/Why Raised
STANDARD.TIME_ERROR	Raised if the elapsed time does not fit within the range of type STANDARD.DURATION

3.3.9.1.10.2 CONVERTED_TIME (FUNCTION BODY) UNIT DESIGN

This function converts an input time to a local time (i.e., converts it to the "local time zone"). A local time is defined as the difference between the input time and the internal reference time.

3.3.9.1.10.2.1 REQUIREMENTS ALLOCATION

This part partially meets requirement CAMP R046.

3.3.9.1.10.2.2 LOCAL ENTITIES DESIGN

None.

3.3.9.1.10.2.3 INPUT/OUTPUT

FORMAL PARAMETERS:

The following table describes this part's formal parameters:

Name	Type	Mode	Description
Clock_Time	CALENDAR.TIME	In	Time to be coverted to a local time

3.3.9.1.10.2.4 LOCAL DATA

None.

3.3.9.1.10.2.5 PROCESS CONTROL

Not applicable.

3.3.9.1.10.2.6 PROCESSING

The following describes the processing performed by this part:

```
function Converted_Time (Clock_Time : in CALENDAR.TIME)
                                return DURATION is

begin

   return Clock_Time - Reference_Time;

end Converted_Time;
```

3.3.9.1.10.2.7 UTILIZATION OF OTHER ELEMENTS

The following library units were previously with'd and are visible to this part:

1. Calendar

UTILIZATION OF EXTERNAL ELEMENTS:

Data types:

The following table summarizes the external types required by this part:

Name	Type	Source	Description
TIME	private	CALENDAR	Implementation-dependent representation of time
DURATION	fixed	STANDARD	Represents a length of time

UTILIZATION OF OTHER ELEMENTS IN TOP LEVEL COMPONENT:

The following tables describe the elements used by this part but defined elsewhere in the parent top level component:

Data objects:

The following table summarizes the objects required by this part and defined in the package body of Clock_Handler:

Name	Type	Value	Description
Reference_Time	CALENDAR.TIME	N/A	Internal reference clock maintained by this part

3.3.9.1.10.2.8 LIMITATIONS

The following table describes the exceptions propagated by this part:

Name	When/Why Raised
STANDARD. TIME_ERROR	Raised if the elapsed time does not fit within the range of type STANDARD.DURATION

3.3.9.1.10.3 RESET_CLOCK (PROCEDURE BODY) UNIT DESIGN

This procedure effectively zeroes out the internal clock by setting the internal reference time equal to the system time.

3.3.9.1.10.3.1 REQUIREMENTS ALLOCATION

This part partially meets requirement CAMP R046.

3.3.9.1.10.3.2 LOCAL ENTITIES DESIGN

None.

3.3.9.1.10.3.3 INPUT/OUTPUT

None.

3.3.9.1.10.3.4 LOCAL DATA

None.

3.3.9.1.10.3.5 PROCESS CONTROL

Not applicable.

3.3.9.1.10.3.6 PROCESSING

The following describes the processing performed by this part:

```
procedure Reset_Clock is

begin

   Reference_Time := CALENDAR.CLOCK;

end Reset_Clock;
```

3.3.9.1.10.3.7 UTILIZATION OF OTHER ELEMENTS

The following library units were previously with'd and are visible to this part:

1. Calendar

UTILIZATION OF EXTERNAL ELEMENTS:

Subprograms and task entries:

The following table summarizes the external subroutines and task entries required by this part:

Name	Type	Source	Description
Clock	Function	Calendar	Returns the internal system time

Data types:

The following table summarizes the external types required by this part:

Name	Type	Source	Description
TIME	private	CALENDAR	Implementation-dependent representation of time
DURATION	fixed	STANDARD	Represents a length of time

UTILIZATION OF OTHER ELEMENTS IN TOP LEVEL COMPONENT:

The following tables describe the elements used by this part but defined elsewhere in the parent top level component:

Data objects:

The following table summarizes the objects required by this part and defined in the package body of Clock_Handler:

Name	Type	Value	Description
Reference_Time	CALENDAR.TIME	N/A	Internal reference clock maintained by this part

3.3.9.1.10.3.8 LIMITATIONS

None.

3.3.9.1.10.4 SYNCHRONIZE_CLOCK (PROCEDURE BODY) UNIT DESIGN

This procedure effectively sets the internal clock to a user-specified time. It does this by setting the reference time to a system (CALENDAR) time - the desired time. By default, the system time used is CALENDAR.CLOCK by the user may supply his own "system" time.

3.3.9.1.10.4.1 REQUIREMENTS ALLOCATION

This part partially meets requirement CAMP R046.

3.3.9.1.10.4.2 LOCAL ENTITIES DESIGN

None.

3.3.9.1.10.4.3 INPUT/OUTPUT

FORMAL PARAMETERS:

The following table describes this part's formal parameters:

Name	Type	Mode	Description
New_Time	STANDARD. DURATION	In	Time to which the internal clock should be set
Clock_Time	CALENDAR.TIME	In	System time

3.3.9.1.10.4.4 LOCAL DATA

None.

3.3.9.1.10.4.5 PROCESS CONTROL

Not applicable.

3.3.9.1.10.4.6 PROCESSING

The following describes the processing performed by this part:

```
procedure Synchronize_Clock
            (New_Time   : in STANDARD.DURATION;
             Clock_Time : in CALENDAR.TIME := CALENDAR.CLOCK) is

begin

   Reference_Time := Clock_Time - New_Time;

end Synchronize_Clock;
```

3.3.9.1.10.4.7 UTILIZATION OF OTHER ELEMENTS

The following library units were previously with'd and are visible to this part:

1. Calendar

UTILIZATION OF EXTERNAL ELEMENTS:

Data types:

The following table summarizes the external types required by this part:

Name	Type	Source	Description
TIME	private	CALENDAR	Implementation-dependent representation of time
DURATION	fixed	STANDARD	Represents a length of time

UTILIZATION OF OTHER ELEMENTS IN TOP LEVEL COMPONENT:

The following tables describe the elements used by this part but defined elsewhere in the parent top level component:

Data objects:

The following table summarizes the objects required by this part and defined in the package body of Clock_Handler:

Name	Type	Value	Description
Reference_Time	CALENDAR.TIME	N/A	Internal reference clock maintained by this part

3.3.9.1.10.4.8 LIMITATIONS

None.

3.3.9.1.10.5 ELAPSED_TIME (FUNCTION BODY) UNIT DESIGN

This function returns the time since the last call to this function. The first call to this function will result in the time since the package was elaborated. This function is not affected by calls to Reset_Clock or Synchronize_Clock.

3.3.9.1.10.5.1 REQUIREMENTS ALLOCATION

This part partially meets requirement CAMP R046.

3.3.9.1.10.5.2 LOCAL ENTITIES DESIGN

None.

3.3.9.1.10.5.3 INPUT/OUTPUT

None.

3.3.9.1.10.5.4 LOCAL DATA

Data objects:

The following table describes the data objects maintained by this part:

Name	Type	Value	Description
Answer	STANDARD.DURATION	N/A	Amount of time which has elapsed since the last call to this function
New_Time	CALENDAR.TIME	N/A	System time

3.3.9.1.10.5.5 PROCESS CONTROL

Not applicable.

3.3.9.1.10.5.6 PROCESSING

The following describes the processing performed by this part:

```
   function Elapsed_Time return STANDARD.DURATION is

--    ---------------------
--    --declaration section
--    ---------------------

      Answer   : STANDARD.DURATION;
      New_Time : CALENDAR.TIME := CALENDAR.CLOCK;

-- -----------------------------
-- --begin function Elapsed_Time
-- -----------------------------

   begin

      Answer           := New_Time - Time_Last_Called;
      Time_Last_Called := New_Time;

      return Answer;

   end Elapsed_Time;
```

3.3.9.1.10.5.7 UTILIZATION OF OTHER ELEMENTS

The following library units were previously with'd and are visible to this part:

1. Calendar

UTILIZATION OF EXTERNAL ELEMENTS:

Subprograms and task entries:

The following table summarizes the external subroutines and task entries required by this part:

Name	Type	Source	Description
Clock	Function	Calendar	Returns the internal system time

Data types:

The following table summarizes the external types required by this part:

Name	Type	Source	Description
TIME	private	CALENDAR	Implementation-dependent representation of time
DURATION	fixed	STANDARD	Represents a length of time

UTILIZATION OF OTHER ELEMENTS IN TOP LEVEL COMPONENT:

The following tables describe the elements used by this part but defined elsewhere in the parent top level component:

Data objects:

The following table summarizes the objects required by this part and defined in the package body of Clock_Handler:

Name	Type	Value	Description
Time_Last_Called	CALENDAR.TIME	N/A	Last time the Elapsed Time function was called

3.3.9.1.10.5.8 LIMITATIONS

The following table describes the exceptions propagated by this part:

Name	When/Why Raised
STANDARD. TIME_ERROR	Raised if the elapsed time does not fit within the range of type STANDARD.DURATION

(This page left intentionally blank.)

```
package body Clock_Handler is

   use CALENDAR;

-- --------------------
-- --local declarations
-- --------------------

   Reference_Time       : CALENDAR.Time := CALENDAR.Clock;
   Time_Last_Called     : CALENDAR.Time := CALENDAR.Clock;

pragma PAGE;
   function Current_Time return DURATION is

   begin

      return CALENDAR.Clock - Reference_Time;

   end Current_Time;

pragma PAGE;
   function Converted_Time (Clock_Time : in CALENDAR.Time)
                                  return DURATION is

   begin

      return Clock_Time - Reference_Time;

   end Converted_Time;

pragma PAGE;
   procedure Reset_Clock is

   begin

      Reference_Time := CALENDAR.Clock;

   end Reset_Clock;

pragma PAGE;
   procedure Synchronize_Clock
               (New_Time   : in STANDARD.DURATION;
                Clock_Time : in CALENDAR.Time := CALENDAR.Clock) is

   begin

      Reference_Time := Clock_Time - New_Time;

   end Synchronize_Clock;

pragma PAGE;
   function Elapsed_Time return STANDARD.DURATION is

--    --------------------
--    --declaration section
--    --------------------
```

```
      Answer   : STANDARD.DURATION;
      New_Time : CALENDAR.Time := CALENDAR.Clock;

-- ----------------------------
-- --begin function Elapsed_Time
-- ----------------------------

   begin

      Answer           := New_Time - Time_Last_Called;
      Time_Last_Called := New_Time;

      return Answer;

   end Elapsed_Time;

end Clock_Handler;
```

4 (NOT USED)

(This page intentionally left blank.)

5 (NOT USED)

(This page intentionally left blank.)

6 NOTES

This paragraph does not apply to this DDD.

(This page intentionally left blank.)

SUPPLEMENTARY

INFORMATION

DEPARTMENT OF THE AIR FORCE
WRIGHT LABORATORY (AFSC)
EGLIN AIR FORCE BASE, FLORIDA, 32542-5434

ERRATA
AD-B120259

REPLY TO ATTN OF: MNOI

13 Feb 92

SUBJECT: Removal of Distribution Statement and Export-Control Warning Notices

TO: Defense Technical Information Center
ATTN: DTIC/HAR (Mr William Bush)
Bldg 5, Cameron Station
Alexandria, VA 22304-6145

1. The following technical reports have been approved for public release by the local Public Affairs Office (copy attached).

	Technical Report Number	AD Number
1.	88-18-Vol-4	ADB 120 251
2.	88-18-Vol-5	ADB 120 252
3.	88-18-Vol-6	ADB 120 253
4.	88-25-Vol-1	ADB 120 309
5.	88-25-Vol-2	ADB 120 310
6.	88-62-Vol-1	ADB 129 568
7.	88-62-Vol-2	ADB 129 569
8.	88-62-Vol-3	ADB 129-570
9.	85-93-Vol-1	ADB 102-654
10.	85-93-Vol-2	ADB 102-655
11.	85-93-Vol-3	ADB 102-656
12.	88-18-Vol-1	ADB 120 248
13.	88-18-Vol-2	ADB 120 249
14.	88-18-Vol-7	ADB 120 254
15.	88-18-Vol-8	ADB 120 255
16.	88-18-Vol-9	ADB 120 256
17.	88-18-Vol-10	ADB 120 257
18.	88-18-Vol-11	ADB 120 258
19.	88-18-Vol-12	ADB 120 259

2. If you have any questions regarding this request call me at DSN 872-4620.

Lynn S. Wargo

LYNN S. WARGO
Chief, Scientific and Technical
Information Branch

1 Atch
AFDTC/PA Ltr, dtd 30 Jan 92

ERRATA

DEPARTMENT OF THE AIR FORCE
HEADQUARTERS AIR FORCE DEVELOPMENT TEST CENTER (AFSC)
EGLIN AIR FORCE BASE, FLORIDA 32542-6000

REPLY TO
ATTN OF: PA (Jim Swinson, 882-3931) 30 January 1992

SUBJECT: Clearance for Public Release

TO: WL/MNA

The following technical reports have been reviewed and are approved for public release: AFATL-TR-88-18 (Volumes 1 & 2), AFATL-TR-88-18 (Volumes 4 thru 12), AFATL-TR-88-25 (Volumes 1 & 2), AFATL-TR-88-62 (Volumes 1 thru 3) and AFATL-TR-85-93 (Volumes 1 thru 3).

Virginia N. Pribyla

VIRGINIA N. PRIBYLA, Lt Col, USAF
Chief of Public Affairs

AFDTC/PA 92-039